THE UNIVERSITY
of EDINBURGH

COOPERATIVE AUTO-TUNING OF PARALLEL SKELETONS

ALEXANDER JAMES COLLINS

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2015

ABSTRACT

Improving program performance through the use of multiple homogeneous processing elements, or cores, is common-place. However, these architectures increase the complexity required at the software level. Existing work is focused on optimising programs that run in isolation on these systems, but ignores the fact that, in reality, these systems run multiple parallel programs concurrently with programs competing for system resources. In order to improve performance in this shared environment, cooperative tuning of multiple, concurrently running parallel programs is required. Moreover, the set of programs running on the system – the system workload – is dynamic and rapidly changing. This makes cooperative tuning a challenge, as it must react rapidly to changes in the system workload.

This thesis explores the scope for performance improvement from cooperatively tuning skeleton parallel programs, and techniques that can be used to cooperatively auto-tune parallel programs. Parallel skeletons provide a clear separation between algorithm description and implementation, and provide tuning knobs that the system can use to make high-level changes to a programs implementation. This work is in three parts: (*i*) how many threads should be allocated to each program running on the system, (*ii*) on which cores should a programs threads be executed and (*iii*) what values should be chosen for high-level parameters of the parallel skeletons. We demonstrate that significant performance improvements are available in each of these areas, compared to the current state-of-the-art.

LAY SUMMARY

Improving program performance by splitting the work between multiple processors is common-place. However, this approach increases the complexity of the software required. Existing work is focused on optimising programs that run on their own on these systems, but ignores the fact that, in reality, these systems run multiple parallel programs at the same time with each program competing for system resources. In order to improve performance in this shared environment, cooperative tuning of multiple, parallel programs that are running at the same time is required. Moreover, the set of programs running on the system varies over time and changes rapidly. This makes this cooperative tuning a challenge, as it must react to these changes.

This thesis explores the scope for performance improvement from cooperatively tuning programs, and techniques that can be used to cooperatively these programs. This work is in three parts: (*i*) how many parts should each program be split into when run on the system, (*ii*) on which processors should a programs parts be executed and (*iii*) how should the way each part executes its work be modified to get best performance. We demonstrate that significant performance improvements are available in each of these areas, compared to the current state-of-the-art.

ACKNOWLEDGMENTS

I would like to thank my academic supervisors, Murray Cole and Chris Fensch for their support over the past years, and Tim Harris at Oracle Labs for his collaboration on Chapter 5. Their advice and guidance have been invaluable.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Alexander James Collins, 8 October, 2015

RELATED PUBLICATIONS

Parts of this thesis have been published in the proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), June 2015.

CONTENTS

1

INTRODUCTION

Relentless improvements in processor performance have historically come from increasing processor frequency and more sophisticated micro-architecture designs. This progress has slowed as processor designs reach physical limits, such as a limited capacity to remove heat (Sutter and Larus, 2005). Using multiple homogeneous processing elements, or cores, is now a focus for industry and research as it is a promising avenue to continue improving processor performance (Asanovic et al., 2006).

To utilise multiple cores, parts of the program must be executed in parallel. However, this increases the complexity required at the software level to obtain optimal performance. This is because the performance relies heavily on efficient utilisation of several cores. In contrast to an efficient sequential implementation, programs now need to address issues including parallel decomposition, work scheduling, resource allocation and inter-core communication.

Similarly to the single-core era, additional complexity is introduced by executing multiple parallel programs at the same time on these multi-core systems. The resource sharing that this causes means that the programs will have a performance impact on one another. Poor implementation decisions made by one program can lead to interference for shared resources between programs and harm overall system performance. Therefore, in order to achieve optimal performance, programs need to consider the presence of other programs running on the system – the *system workload*. Unlike the complexities introduced by the use of multi-core hardware, system workload is dynamic and rapidly changing. This dynamism is caused by new programs starting execution, completing execution, or exhibiting phase changes in their behaviour at unpredictable points. To maintain optimal performance, parallel programs need to adapt to these changes in system workload dynamically at runtime.

Due to this increase in complexity, achieving optimal performance via manual implementation of parallel programs requires significant human effort and expertise. This has led to parallel abstractions that aim to hide this complexity from the programmer without introducing significant or unpredictable overheads (Cole, 2004).

Parallel skeletons, also known as algorithmic skeletons, are one such abstraction for parallel computing (Cole, 1989). They provide an application programmer with a

collection of skeletons, or templates, each of which implements a standard algorithmic technique. Common skeletons include task farms, pipelines, divide and conquer, and data-parallel map and reduce (González-Vélez and Leyton, 2010). A program is implemented by nesting and composing appropriate skeletons, and parameterising them with application specific sequential code.

Parallel frameworks, such as this, provide a clear separation between algorithm description and implementation: *what* the algorithm should do is specified at a high-level, by the choice of skeletons; and the compiler and runtime are free to choose *how* the algorithm should be implemented. This allows the application programmer to focus on solving their particular problem, without worrying about low-level implementation details or performance. It also enables the compiler and runtime to make implementation decisions based on high-level algorithm structure.

However, existing approaches to parallel abstraction ignore the issue of optimising multiple, concurrently running programs. Previous work is focused on either optimising parallel programs individually, where there is just one program executing in isolation (Ansel et al., 2009; Wang and O'Boyle, 2010; Christen et al., 2011; Dastgeer et al., 2011), or optimising multiple programs that have been decomposed into a common low-level representation (Intel, 2012; Dagum and Menon, 1998; OpenMP, 2015). Both approaches ignore high-level algorithm information that is likely beneficial for program optimisation.

We therefore need a system that can achieve optimal performance across a set of concurrently executing programs running on multi-core hardware, by adapting their implementation to the changing system workload. We call this *cooperative autotuning*. The clear separation between *what* a program should do and *how* it should do it, provided by parallel frameworks, can be exploited to achieve this goal. High level algorithm information can be used to inform the automatic tuning process. This thesis explores the scope for performance improvement from cooperatively tuning parallel programs, devising techniques to improve overall system performance.

The rest of this chapter is structured as follows. Section 1.1 discusses scenarios that benefit from cooperative tuning, due to the dynamic nature of their system workload, Section 1.2 describes three cooperative tuning challenges that are explored in this work and Section 1.3 summarises the contributions made. Section 1.4 provides an outline for the remainder of the thesis.

1.1 DYNAMIC SYSTEM WORKLOAD

There are many situations where system workload continuously evolves and it is in these scenarios that cooperative tuning is essential. The following examples describe real-world scenarios where cooperative tuning of parallel programs is beneficial, specifically due to the dynamically changing nature of the system workload.

MULTI-CORE WORKSTATION Consider a multi-core CPU running a program that is fully utilising all of the machine's cores. If a second program starts executing, the system workload changes. The best overall system performance is now achieved using a subset of the cores for each program. This requires the currently running program to dynamically adjust the number of threads it is executing. This switch needs to happen at runtime, in response to this unpredictable change in the system workload. This demonstrates the need for cooperative tuning, even in this small scale example. Dynamic workloads such as this, where a changing mix of programs are run concurrently, are commonplace on these systems (Asanovic et al., 2006).

IMAGE PROCESSING ON A MOBILE PLATFORM Dynamic workloads are also common on mobile platforms (Kim et al., 2013). For example, a mobile application that provides continuous panorama stitching of images captured from a camera requires motion detection to determine when an image should be captured and, once captured, the new image needs to be stitched onto the panorama. These two components need to run concurrently. The motion detection component runs continuously, and the image stitching component runs occasionally, leading to bursty system workload. Cooperatively tuning these two components maintains the responsiveness of the system, in the face of the dynamic nature of the workload.

CLOUD AND UTILITY COMPUTING Cloud and utility computing is a much larger scale example of where cooperative tuning is beneficial. The aim of cloud and utility computing is to provide computing as a service. Many customer applications are run across a large cluster of machines. In order to make the most efficient use of the hardware, the cloud service provider needs to maximise the throughput of these multi-application nodes. The applications run are varied and the system workload evolves rapidly over time (Feinberg, 2006), therefore cooperative tuning is essential to maximise the system throughput of this dynamic mix of workloads.

1.2 THREE CHALLENGES FOR COOPERATIVE AUTO-TUNING

There are many challenges involved with cooperatively auto-tuning multiple, concurrently running parallel programs for shared memory multi-core systems. This thesis explores three of these challenges. Other potential challenges in this field are discussed in Section 7.3.1.

The first challenge is how many *software threads* each program running on the system should use. Shared memory multi-core systems consist of multiple processing elements, or cores, each of which can execute an independent stream of instructions, or *hardware thread*. Programs create software threads which are scheduled onto these hardware threads for execution. Choosing the number of software threads to create has a significant effect on performance, due to issues including the scalability of the program and choices made by other programs running on the system. The second challenge is which hardware threads, or cores, these software threads should be executed on. For example, on a multi-socket system this impacts performance as it affects inter-socket communication overheads and contention for resources within sockets. Finally, the third challenge addressed in this thesis is how to tune high-level parameters that are exposed by parallel frameworks. In this work, we explore a high-level parameter that controls the granularity of computation in Intel's Threading Building Blocks framework (Intel, 2012).

These three challenges are discussed in more detail in the following sections. This thesis addresses each of these challenges individually in Chapter 4, Chapter 5 and Chapter 6 respectively. It is likely that these challenges are sometimes inter-dependent. However, addressing the implied trade-offs is beyond the scope of this thesis.

1.2.1 *Challenge One: Tuning Thread Count*

Parallel programs execute multiple instruction streams, called *software threads*. These threads are mapped to the underlying hardware by the Operating System (OS) scheduler. Each processing unit, or core, on the system can execute a fixed number of these instruction streams at a time, called *hardware threads*. Typically there are many more software threads than hardware threads, and the OS scheduler is responsible for giving a fair share of execution time to each software thread running on the system.

The choice of how many software threads to spawn is left to the individual programs. Typical OS schedulers simply schedule the threads that they are given, and do

not have direct control over how many threads programs will spawn. Typical parallel applications either spawn as many threads as there are cores, or spawn a manually tuned number of threads. Importantly, this choice is not based on the number of threads that are currently executing on the system.

This arrangement causes several issues:

UNDER-UTILISATION If there are fewer software threads than hardware threads, the available hardware resources are under-utilised. To improve system performance, programs need to spawn sufficiently many threads to ensure that each hardware thread has useful work that it can perform.

OVER-SUBSCRIPTION This occurs when there are many more software threads than hardware threads. Programs typically choose their number of threads in order to avoid under-utilisation. Spawning as many software threads as there are hardware threads is a common approach. However, this ignores the threads spawned by other programs running on the system. This means that when multiple programs are executing there are likely many more software threads than hardware threads. This leads to the OS scheduler performing context switching between threads to maintain fairness. However this introduces overheads which harm overall system performance. For example, repeated context switching between different threads can lead to high cache miss rates, as memory used by one thread is evicted from the cache when another thread performs work.

SCALABILITY Program performance does not necessarily scale linearly as the number of threads increases. For a program to scale 'well' its performance needs to increase as the number of threads increases. However, many programs achieve diminishing improvements in performance as the number of threads is increased. For example, this could be due to the increase in inter-thread communication overhead as the number of threads is increased. Moreover, some programs scale poorly. In these cases, program performance decreases as the number of threads increases. For example, this could be due to thread synchronisation overheads exceeding the amount of useful work that the threads are completing.

The challenge is to choose the best number of threads for every program running on the system, such that optimal system performance is achieved. This decision needs to take the amount of hardware resources available and scalability of the programs into account. Moreover, this tuning needs to be performed dynamically, due to the dynamic nature of the system workload as discussed in Section 1.1.

1.2.2 *Challenge Two: Tuning Thread Placement*

In addition to choosing how many threads each program should spawn, a decision of where each thread should be executed also needs to be made. Shared memory multi-core multi-socket systems consist of several groups of cores. Each socket is distinct, with its own cache hierarchy, memory controller and main memory. Sockets are connected to one another via an inter-socket interconnect. This interconnect is relatively slow compared to the intra-socket communication between cores.

This spatial scheduling problem of where to execute threads has a significant effect on performance, for many reasons, including the following:

INTER-SOCKET COMMUNICATION Inter-socket communication uses an inter-connect network to relay messages. Communication via this interconnect can introduce large overheads in program execution, as accessing memory in caches on different sockets via the inter-socket interconnect incurs a much higher latency than accessing socket-local caches. For example, Intel's QuickPath Interconnect (Intel, 2009) can introduce over 100 nanoseconds of latency to cache accesses, compared to at most 70 nanoseconds for on-chip caches (Molka et al., 2009). The thread to core schedule needs to be chosen such that inter-socket communication is minimised.

CACHE UTILISATION AND LOCALITY Threads should ideally always be executed on the same core. This helps with memory caching, as it increases data locality. Data cached for the thread is likely still resident in the lower level caches of the memory hierarchy, which provide lower access latency and higher bandwidth.

SEGMENTATION OF RESOURCES Each socket contains a complete processor, consisting of multiple cores, and has its own hardware resources such as a memory controller and off-chip memory. Therefore, the hardware resources are segmented between multiple sockets. Threads need to be scheduled to cores such that the hardware resources of each socket are fully utilised. For example, each socket has its own memory controller and attached main memory. Threads should be allocated to cores such that the memory pressure on each socket is balanced.

These issues are interrelated and need to be addressed together. For example, placing threads from the same program on the same socket may reduce inter-socket communication, however it would also serve to reduce the amount of cache available

to the program. This has the potential to harm performance: the reduction in inter-socket communication overheads may be outweighed by the higher capacity miss rate in the caches when performing memory accesses.

Typical OS schedulers choose where each thread is run using a simple heuristic: execute the thread where it has been executing previously. However, this does not take into account the segmentation of resources, and does not necessarily lead to resource utilisation being balanced amongst the sockets.

A spatial scheduler for multi-socket systems that takes these issues into account is required. Moreover, the scheduler needs to be dynamic and adaptive due to the dynamic nature of the system workload as discussed in Section 1.1.

1.2.3 *Challenge Three: Tuning Parallel Framework Parameters*

In addition to the choice of how many threads to spawn and where to schedule them, parallel abstractions also provide a range of implementation parameters whose values have a direct impact on performance.

For example, Intel's Threading Building Blocks framework (Intel, 2012) contains a *grain size* parameter which affects the granularity of the computation performed by parallel programs running on the system. A smaller grain size causes a large number of small parallel tasks to be executed, whereas a larger grain size will result in a smaller number of large tasks. A small grain size provides more scope for parallel execution, however it will increase overheads such as inter-thread communication, thus impacting system performance.

Similarly to thread count and thread placement, we require automatic tuning of implementation parameters such as the grain size parameter in TBB. Implementation parameters directly affect the behaviour of parallel programs, and their resource usage characteristics, and therefore have a direct impact on the performance of parallel programs. Carefully tuning them is therefore an important task when trying to optimise overall system performance.

These parameters are also likely to affect other programs running on the system. These parameters therefore need to be tuned in a cooperative manner.

1.3 CONTRIBUTIONS

This thesis addresses the three cooperative tuning challenges described in the previous section. Techniques for auto-tuning the thread count, thread placement and choice of high-level parameters affecting the implementation are explored. Thread count tuning is performed dynamically at runtime, using scalability information collected for each program a priori. Thread placement tuning is performed using a heuristic driven scheduler. The heuristic uses memory pressure to allocate program's threads to sockets. Finally, automatically tuning the grain size implementation parameter in Intel's Threading Building Blocks is explored.

The main contributions of this thesis are:

- An online, adaptive scheduler that chooses how many threads each parallel program running on the system should use, with the aim of maximising overall system performance. Thread counts are chosen based on scalability information for the programs, collected using an offline training phase. The approach dynamically adapts its decisions runtime, in response to changes in the system workload. Runtime overheads are reduced, by using static program information to determine when the system workload changes. Our scheduler improves overall system performance, compared to manually tuned benchmarks written using the OpenMP (Dagum and Menon, 1998) parallel framework. This work is presented in Chapter 4.

- A heuristic and scheduler to schedule program threads to sockets is developed. The effect of different program-to-core mappings on performance is explored, revealing the performance degradation caused by ignoring the presence of separate sockets. An online adaptive scheduler that reduces interference and resource contention on shared memory multi-core multi-socket systems is developed. Its aim is to balance the pressure on the memory system of each socket, which results in improvements in system performance. This online scheduling approach is compared against two competing approaches: Callisto (Harris et al., 2014) and OpenMP (Dagum and Menon, 1998), and our approach achieves improvement in system performance. The system performance when using the dynamic scheduler is also compared to an offline static approach using the same heuristic. It is shown that the dynamic approach improves system performance. This work is presented in Chapter 5.

- An exploration of the optimisation space of the grain size implementation parameter that is part of Intel's Threading Building Blocks, and steps towards devising a tuning heuristic. An oracle study of the optimisation space is performed, comparing the best, worst and default performance achieved when varying TBB's grain size parameter. A heuristic is devised which uses TBB performance statistics to tune the grain size parameter, and this heuristic is evaluated using the oracle study data set. This work is presented in Chapter 6.

1.4 THESIS OUTLINE

The remainder of this thesis is structured as follows.

CHAPTER 2 gives background on the areas explored in this thesis. It starts with a description of shared memory multi-core systems, followed by a summary of standard techniques for auto-tuning programs. It then gives background details of parallel skeletons and the libraries and frameworks used in this work. It finishes with a discussion of the pitfalls of, and techniques for, measuring real-world performance on shared memory multi-core systems.

CHAPTER 3 explores related work, providing a critical analysis and comparison with the work presented in this thesis. First, it describes existing parallel skeleton frameworks that perform single-program auto-tuning. It then covers work related to the three cooperative tuning challenges: (*i*) cooperatively tuning the number of threads used by programs, (*ii*) cooperatively tuning the placement of threads to cores in multi-socket systems and (*iii*) cooperatively tuning high-level skeleton parameters and adapting their implementation.

CHAPTER 4 explores cooperative auto-tuning of program thread counts for programs run in a shared environment. A scheduler is developed, which uses scalability information collected in an offline training phase to decide how many threads each program on the system should use, with the aim of improving overall system performance. It dynamically adapts to changes in system workload, and these points in execution are found with low-overhead, by utilising static program information. Compared to manually tuned implementations of the benchmarks, this dynamic scheduler approach improves system performance, measured using ANTT and STP metrics.

CHAPTER 5 explores cooperative auto-tuning of the placement of a programs threads on a multi-socket machine. A heuristic-driven dynamic scheduler that allocates program threads to sockets to reduce interference and resource contention on shared memory multi-core multi-socket systems is developed. Compared to competing approaches, this dynamic scheduler approach achieves improvement in system performance, measured using ANTT and STP metrics.

CHAPTER 6 explores cooperative auto-tuning of high-level parallel parameters, specifically the grain size parameter in Intel's Threading Building Blocks parallel framework (Intel, 2012). A tuning heuristic is devised which improves overall system performance compared to the default grain size setting in TBB.

CHAPTER 7 concludes the thesis with a summary and discussion of the contributions of this work, and explores directions for potential future work.

2

BACKGROUND

This chapter provides the background knowledge necessary to understand the contributions made in this thesis. It begins in Section 2.1 which describes the multi-program, multi-core, shared memory systems on which this work explores cooperative auto-tuning, and the types of programs that they execute. Section 2.2 then discusses a variety of auto-tuning techniques, including the applicability, benefits and drawbacks of each. This is followed by a discussion of parallel abstractions in Section 2.3, including parallel skeletons and how they are amenable to auto-tuning, and two commonly used parallel frameworks: OpenMP and Intel's Threading Building Blocks. Section 2.4 discusses performance metrics for measuring and comparing the performance of multiple programs, and the techniques and methods necessary to perform reliable and accurate performance measurements in this context. Finally, the chapter concludes with a summary in Section 2.5.

2.1 MULTI-PROGRAM SYSTEMS

This work on cooperative auto-tuning is performed in the context of *multi-program multi-core* systems. These architectures execute multiple programs concurrently, each of which consists of multiple threads, using a homogeneous set of processing elements, or *cores*.

Section 2.1.1 and Section 2.1.2 describe the layout and structure of multi-core and multi-socket shared memory systems and the software that runs on them. Section 2.1.3 gives a historical perspective on why we have come to use such systems. Other multi-program systems are discussed in Section 2.1.4. These systems are not explored in the technical work presented in this thesis, but are included to provide a broader picture of the field.

2.1.1 *Shared-Memory Multi-Core Systems*

Shared-memory multi-core systems contain a Central Processing Unit (CPU) that can execute multiple instruction streams at the same time. They do this by providing multiple processing elements, or *cores*. Each core is largely independent, in the sense that it can execute an instruction stream independently from the other cores in the system. However, the cores share memory. This is the mechanism via which cores communicate with one another – by reading and writing to shared regions of the memory.

A multi-core processor also contains a cache hierarchy to improve memory bandwidth and latency. Each core typically has its own private Level 1 instruction cache and separate Level 1 data cache (L1I and L1D), and private Level 2 cache (L2). A shared Level 3 (L3) cache is also typical, allowing cores to communicate with one another without needing to go all the way to main memory. Multi-core processors are manufactured as a single die containing all of the cores. Main memory is stored off-chip, connected to the memory hierarchy by a fast interconnect.

Typically, a software scheduler in the Operating System schedules software threads onto the available cores. Each core is capable of running at least one hardware thread. On architectures that support Simultaneous Multi-Threading, often referred to as hyper-threading, each core is capable of running more than one hardware thread.

For example, Figure 1 shows a schematic diagram for the i7-3820 CPU used for the experiments in Chapter 6. This CPU consists of 8 cores, each of which has a private L1I, L1D and L2, and share an L3 and main memory with the other cores in the CPU. Each core is capable of running up to 2 hardware threads.

Multi-core systems can also either be homogeneous, where the architecture of each core is identical, or heterogenous where different cores may provide different capa-

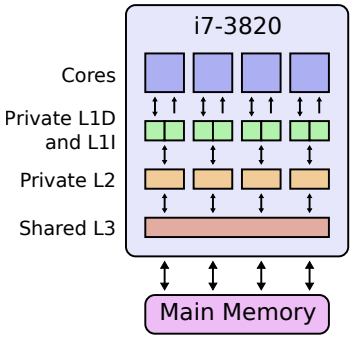

Figure 1: Example of a multi-core processor, showing a schematic for Intel's i7-3860 quad-core CPU, including the cores and memory hierarchy.

bilities or behaviour. For example, cores could be superscalar, VLIW, contain vector units, provide SIMD extensions or offer Simulataneous Multi-Threading. In this work, we focus on heterogenous multi-core systems.

2.1.2 *Multi-Socket Systems*

Multi-socket systems consist of several shared-memory multi-core CPUs, each of which is connected to a different *socket*. These sockets are usually provided by the motherboard as a set of physical pins to which a multi-core CPU can be connected. The motherboard also provides, for each socket, a separate memory controller and associated pins to connect main memory modules.

Also provided by the motherboard is a fast interconnect between the sockets, which allows data to be shared between the cache hierarchies of the separate multi-core CPUs. Unlike shared-memory multi-core CPUs, communication between cores cannot be performed via main memory if the cores are located in separate sockets, due to each socket having a separate memory controller and main memory. Instead cores on different sockets share data via an inter-socket interconnect network, such as Intel's QuickPath. This allows data from the main memory of one socket to be cached in the cache hierarchy of another socket.

For example, Figure 2 shows a schematic diagram for a dual-socket system with a pair of Xeon E5-2660 multi-core processors. Each multi-core processor contains 8 cores, each with its own private L1I, L1I and L2. Each multi-core CPU then contains an L3 which is shared between the cores in each socket, and each is connected to it's own dedicated memory controller and main memory.

Tuning thread placement for this multi-socket system is explored in Chapter 5.

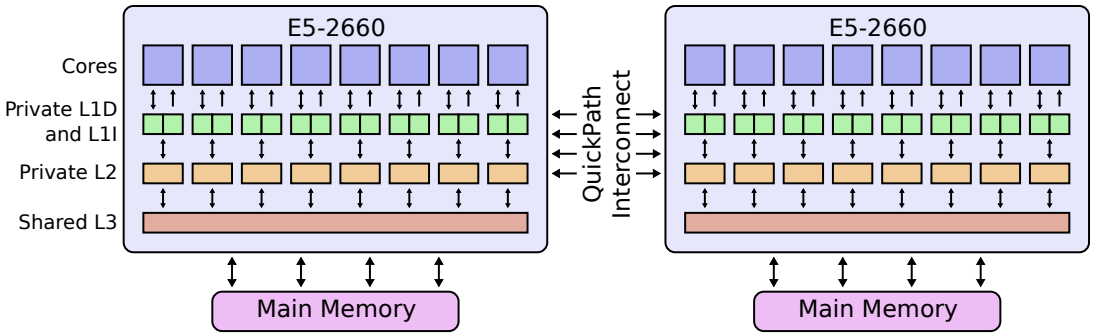

Figure 2: Example of a multi-core multi-socket processor, showing a schematic for a pair of Intel's Xeon E5-2660 CPUs, including the cores, memory hierarchy and inter-socket interconnect.

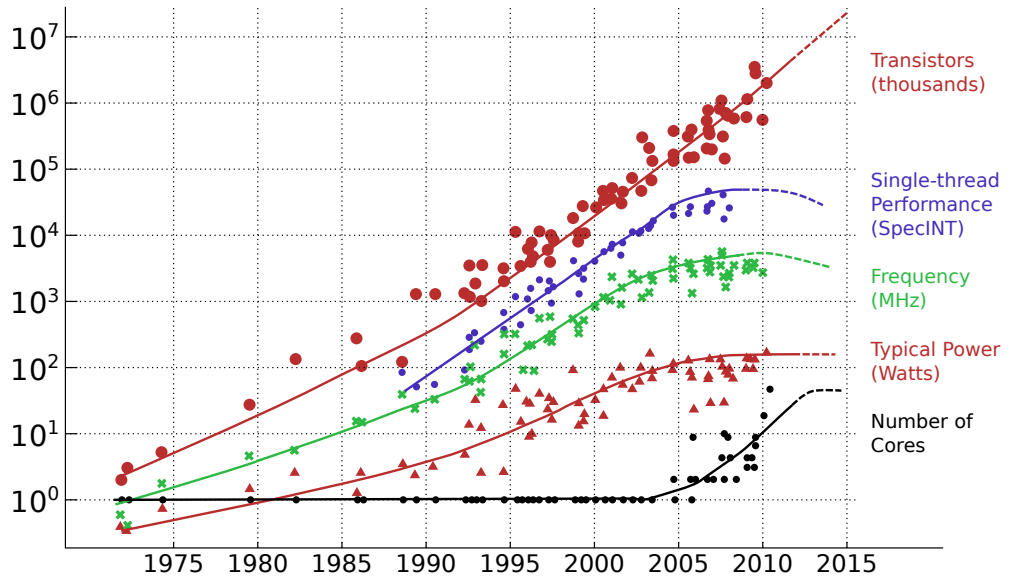

Figure 3: Architectural trends from 1975 – 2015, showing the exponential increase in transistor count, but the failure of performance scaling after 2005. Reproduced from Moore (2011).

2.1.3 *Historical Perspective*

Processor scaling over the last few decades has provided a steady increase in the number of transistors that fit on a single chip – mainly from decreasing process sizes. Until roughly a decade ago, this allowed increasingly complex architectures to be designed and realised, providing steady improvements in performance. (Sutter and Larus, 2005)

These early single processor machines provided improved performance even for legacy codes. Application programmers did not need to concern themselves with the underlying architecture – they could, to a large extent, rely on the next generation of processor architecture providing them with increased performance without needing to modify the software.

However, we have now reached physical limits that have drastically slowed this process scaling, and so increasing the complexity of single processor architectures is no longer an avenue for increasing performance. These trends are shown in Figure 3. The focus has therefore shifted to multi-core systems, which consist of multiple simpler cores. These systems allow programs to run multiple threads concurrently for increased performance. However, changes are now needed at the software level to exploit the available parallelism. As the cores in a multi-core system become simpler and more numerous, existing legacy software may actually run slower. It is no longer

the case that application developers can simply assume that their application will run faster on the next generation of processors.

Performance is not the only driving force behind this increase in concurrency. We also want asynchronous processing, for example to implement responsive user interfaces, which requires concurrent processing.

The story does not end here. The increasing use of cloud computing and virtualisation to provide computing as a service has made running multiple programs at the same time a necessity. This is not a new use case, however optimal performance is important in these scenarios to keep hosting costs to a minimum.

2.1.4 *Alternative Systems*

The following discusses examples of alternative systems on which parallel applications can be run. This thesis focuses on multi-core and multi-socket shared-memory systems, described in Section 2.1.1 and Section 2.1.2, but discussion of these alternative systems is included for completeness.

HETEROGENEOUS CPUS This thesis focuses on multi-core CPUs where each core has identical capabilities and architecture. In contrast, heterogeneous multi-core CPUs contain cores whose architecture and capabilities differ.

For example, ARM's big.LITTLE architecture (Greenhalgh, 2011) combines high-performance and high-energy consuming cores, such as the ARM Cortex-A15, with lightweight lower-performance and more energy efficient cores, such as the ARM Cortex-A7. Depending on the system workload, the high-performance cores can be powered down and threads migrated to the lighter weight cores. This serves to improve energy efficiency for embedded devices where use of a high-performance core is not needed all of the time.

GENERAL PURPOSE GPUS General Purpose Graphics Processor Units (GPGPU) are a fundamentally different design of processor to multi-core CPUs. They consist of a large number of simpler cores. Each core executes the same instruction stream in lock step, but on different data – a single-instruction multiple-data (SIMD) processor. This design makes GPUs suitable for data-parallel tasks such as applications used in image processing.

This SIMD design means that the cores are not efficient at executing code where the control flow path diverges for different inputs data. GPUs are often struc-

tured as several independent SIMD processors, often called Streaming Multiprocessors (SMs), where each SM consists of multiple cores and each SM can execute different a different stream of instructions. GPUs are used in combination with a traditional CPU, which coordinates the execution of programs on the GPU, often called kernels, and instructs the GPU to copy data to and from main memory.

Systems can also be built from multiple GPUs, in a similar way to multi-socket CPUs.

CLUSTER COMPUTING Clusters consist of many computers connected by a local area network. Large scale parallel jobs are run on the cluster, which are divided into smaller tasks to run on each of the machines in the cluster.

Message Passing Interface (MPI) is a commonly used framework for programming these systems, providing abstractions for communication between nodes in the cluster.

INTEL'S MANY INTEGRATED CORES ARCHITECTURE The Xeon Phi, developed by Intel in 2009 (Jeffers and Reinders, 2013), is a CPU that consists of 48 single-core processors connected by a fast interconnect, in a similar manner to nodes in a cloud computing data-center. Each core is distinct, and can communicate point-to-point with any other core via the interconnect. This is an example of their Many Integrated Cores (MIC) architecture, providing a high-degree of parallelism.

HETEROGENEOUS SYSTEMS These systems combine processors and accelerators of different types into a single system. For example the combination of a multi-core CPU with a GPU, or CPU with an Field Programmable Gate Array (FPGA) as an accelerator.

These systems can provide benefits for performance and energy efficiency if the workload being executed is suited to the accelerators that the system contains.

2.2 AUTO-TUNING

Auto-tuning is a technique to automatically improve the performance of programs. A program's implementation and/or characteristics of its runtime are automatically adjusted, or *tuned*, to improve a chosen performance metric.

The space of tunable parameters is typically very large; making it prohibitively expensive to evaluate program performance at every point in the space, especially when performance measurement requires executing the program. This makes exhaustive search of the parameter space impractical, motivating the need for more sophisticated auto-tuning techniques, which are discussed in the following sections.

2.2.1 *Static vs. Online Auto-Tuning*

Auto-tuning techniques can be divided into two types: *static* auto-tuning and *online* auto-tuning. Static tuning approaches tune a program's implementation at compile time, before the program is run. This means that the tuning decisions are fixed at runtime. In contrast, online tuning approaches continuously review and update the tuning decisions at runtime. This allows the program to adapt to dynamic changes in the system, however it can introduce overhead as the tuning decisions require computation whilst the program is running.

In the context of the work presented in this thesis, online tuning approaches are needed, so that the system can adapt to the dynamically evolving system workload. However, static auto-tuning techniques are also covered here for completeness.

2.2.2 *Iterative Compilation*

Iterative compilation, introduced by Bodin et al. (1998) and Kisuki et al. (2000), is a mechanism that provides a trade-off between the time required for compilation and the execution time of the resulting program. In its simplest form, iterative compilation generates multiple versions of a program and measures the performance of each. The version that achieves the best performance is chosen as the output of the compilation.

There are many disadvantages to this technique. Firstly, the auto-tuning is performed at compile time, and so is incapable of dynamically adapting to changes in the system or changes to the program's inputs. Iterative compilation also requires test inputs to be provided, with which the program's performance is evaluated. These test inputs need to be carefully chosen such that they are representative of the inputs encountered when the program is run in a production environment. Finally, many programs have a very large optimisation space, making this exhaustive technique impractical due to the prohibitively long amount of time required to find a good point in the optimisation space.

2.2.3 *Speeding Up Iterative Compilation*

The main disadvantage of iterative compilation is that the compilation phase takes a prohibitively long amount of time to run. This is due to the optimisation space being large, and the measurement of a program's performance taking time. More sophisticated techniques can help reduce the time required for this search, or remove the need for the search entirely.

2.2.3.1 *Search Techniques*

One technique to speed up iterative compilation and make it practical is to reduce the number of performance measurements required at compile time using a more sophisticated search technique. Search techniques developed in the Machine Learning and AI disciplines, including stochastic search techniques Luke (2009), can be used to do this.

However, the applicability of any search technique is dependent on the nature of the optimisation space Bishop (2006). For example, if the space is shaped like an n-dimensional 'bowl', gradient search will perform well. However, gradient search will produce poor results if the optimisation space contains many local minima, or contains discontinuities. Understanding the search space of a program, or a class of programs, is an important consideration when choosing which search technique to employ, and it may not be possible to automate this.

2.2.3.2 *Pruning the Search Space*

Another technique for reducing the number of evaluations is to reduce the size of the search space, by ignoring regions of the space that exhibit bad performance across all programs. For example, Triantafyllis et al. (2003) develop an algorithm for choosing the best compiler optimisation flags. It reduces the search space of flags using an offline training phase. The space is first pruned by a human expert, to remove obviously bad choices. The performance of a set of training programs is then measured exhaustively across the space, and programs that perform similarly for the same configurations are grouped together. Similarity between a new program's performance and those evaluated in the training set is then used to prune the search space, drastically reducing its size.

A disadvantage of this technique is that the initial training phase is very costly. The initial training is also not portable across platforms, as the technique has to be retrained for each new architecture.

2.2.3.3 *Predicting Program Performance*

Many iterative compilation techniques (Aldinucci and Danelutto, 1999; Dastgeer et al., 2011; Triantafyllis et al., 2003) do not actually execute the program to measure its performance. Instead they employ performance prediction tools to estimate the performance of the program at different points in the optimisation space. This reduces the time required to evaluate each point in the search, thus reducing the time required for compilation.

2.2.4 *Machine Learning*

Machine Learning (Bishop, 2006) can be applied to auto-tuning, by employing a learnt model to predict the best optimisation choice for a program. This requires the use of prior knowledge about the programs or system, acquired during an offline training phase at compiler *construction* time.

Firstly, training data is collected from a set of set of training programs. For example, this could be the performance of each program at different points in the optimisation space. This training data is then used to train a model, that provides a mapping from programs to an approximation for the best point in the optimisation space.

This avoids the prohibitively expensive compilation times required by iterative compilation, by moving program performance measurement to an a priori training phase performed at compiler construction time (Thomson, 2009).

Work by Agakov et al. (Agakov et al., 2006) demonstrates the use of machine learning to auto-tune loops in sequential code. They investigate two models: (*i*) a simple model that assumes each optimisation is independent and identically distributed (IID), and (*ii*) a Markov model which includes information about the relationship between different optimisations.

A large set of program features were manually identified, and Principal Components Analysis (Bishop, 2006) used to reduce this set of features to a smaller, but still representative set. The models were then trained by measuring the execution time of a suite of training programs.

Their results show an average performance increase of 34% for 5 evaluations using their simple IID model, compared to a 29% speedup for 50 evaluations using a random search. This shows that machine learning techniques can be effective in reducing the number of program evaluations required whilst maintaining similar speedup.

The effectiveness of machine learning is strongly dependent on the choice of program features, which is often performed manually by a human expert. However, Leather et al. (Leather et al., 2009) develop a method for automatically choosing features. They report that their approach finds 76% of the available program performance, compared to 59% by state-of-the-art machine learning approaches with manual feature choice and 3% by a standard hand-tuned compiler using heuristic optimisations.

The generation of training data is a slow process. Although machine learning avoids performing this expensive task at compile time, it still needs to be performed at compiler construction time. It still requires a representative set of training programs and associated training inputs to be provided, usually by a human expert.

2.2.5 *Scheduler-Based Approaches*

Another technique for auto-tuning is the use of an online scheduler to perform resource allocation and tuning decisions. Schedulers periodically adjust the programs running on the system. For example, the default Linux OS scheduler periodically context switches threads to run on the CPU, in order to balance the processor time that each thread receives. Similarly, a scheduler used to perform auto-tuning will periodically adjust implementation parameters of the programs running on the system to tune their execution and improve performance.

In contrast to iterative compilation and machine learning techniques, schedulers perform these decisions at runtime, allowing them to adapt to changes in the system, such as the variation in system workload. They can also utilise information collected online, for example characteristics about the current state of the system collected from hardware performance counters (discussed in more detail in Section 2.4.2.2).

2.3 PARALLEL ABSTRACTIONS

This section discusses parallel skeletons, how they are amenable to auto-tuning and two commonly used parallel frameworks: OpenMP and Intel's Threading Building Blocks.

2.3.1 *Parallel Skeletons*

Parallel skeletons are an abstraction for parallel computing (Cole, 1989). They are also commonly referred to as *algorithmic skeletons* or *parallel patterns*. They provide an application programmer with a collection of *skeletons* each of which implements a standard algorithmic technique. Common skeletons include scatter, gather, map, task farms, pipelines, divide and conquer, and data-parallel map and reduce (González-Vélez and Leyton, 2010). Some examples of these skeletons are shown in Figure 4.

Skeleton parallel programs are implemented by nesting and composing these skeletons, and parameterising them with application specific sequential code. For example, consider the example program in Listing 1. This simple loop uses conventional, imperative-style code to increment all of the elements in an array. It could instead be expressed using parallel skeleton, using code akin to that in Listing 3.

Manually parallelising and optimising the low-level code version in Listing 1 leads to an increase in program complexity. For example, Listing 2 shows the same code optimised for a system with SIMD capabilities. Furthermore, any manual optimisation choices made are often tied to the target hardware on which the program is run.

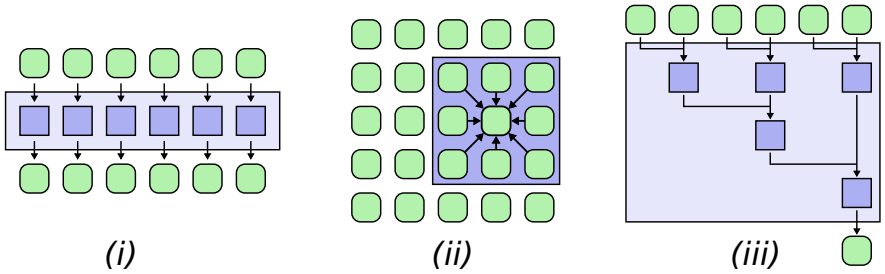

Figure 4: Examples of parallel skeletons: (*i*) a map skeleton that performs an operation for every element in an input array to produce an output array; (*ii*) a stencil skeleton that performs nearest-neighbour computation on a 2-dimensional array; and (*ii*) a reduction skeleton that computes an aggregate value from an input array. Green boxes show elements of the input or output. Blue boxes represent computation and arrows show the flow of data through the algorithm. (McCool et al., 2012)

Listing 1: Increment the elements of an array using low-level code

```
for (int i = 0; i < 1024; i++)
  b[i] = a[i]+1;
```

Listing 2: Increment the elements of an array using low-level code and SIMD extensions

```
for (int i = 0; i < 1024; i += 4)
  b[i:i+4], a[i:i+4], 1);
```

For example, this SIMD example won't work on a systems without support for 4-element wide vector addition. Due to differences in the architecture and capabilities of different systems, this means that a program that has been manually tuned for one system will not necessarily achieve optimal performance on a different system. The program is therefore not *performance portable*. Instead of manual tuning, automatic tuning is required to obtain performance portability. Performing this parallelisation and optimisation automatically on this style of low-level code is challenging, and it is often impossible to auto-tune such low-level codes to generate an optimal program.

In contrast, parallel skeletons provide a clear separation between algorithm description and implementation: *what* the algorithm should do is specified at a high-level, by the choice of skeletons; whereas the compiler and run-time are free to choose *how* the algorithm should be implemented. This allows the application programmer to focus on solving their particular problem, without worrying about low-level implementation details or performance. It also enables the compiler and run-time to make implementation decisions based on high-level algorithm structure provided by the skeletons.

2.3.2 *Auto-tuning Parallel Skeletons*

As parallel skeletons provide a very clear separation between an algorithms description and its implementation, they are amenable to automatic tuning. For example, a skeleton implementation may provide a variety of *tuning knobs* for each skeleton, and varying these does not require modification of the user program. This also enables the tuning to be performed in a platform agnostic way, as optimisation is not performed in the source code.

Listing 3: Increment the elements of an array using a map skeleton

```
map(a, b, [](int x) { return x+1; });
```

Listing 4: Increment the elements of an array using OpenMP

```
#pragma omp parallel for
for (int i = 0; i < 1024; i++)
  b[i] = a[i]+1;
```

For example, the compiler or runtime could switch skeleton implementations depending on the target system. For example using a SIMD implementation if SIMD extensions are available on the system, or if not, using a sequential loop. Another example is a parameter controlling the number of threads to use to perform the computation in parallel.

There is a large body of existing work on auto-tuning skeleton implementation parameters in a single-program context. These works are discussed in related work, Section 3.3.

2.3.3 *OpenMP*

OpenMP is a shared memory parallel programming framework for C and Fortran (Dagum and Menon, 1998; OpenMP, 2015), supporting both task and data parallelism. It provides a set of pragmas that can be used to annotate sections of the program, which provide rich high-level information to an OpenMP enabled compiler so that it can automatically parallelise the code. For example, Listing 4 shows how the vector increment example in Listing 3 can be implemented in OpenMP.

An OpenMP program can also be compiled by an unmodified, sequential C compiler by simply ignoring the OpenMP pragmas. OpenMP also allows rapid parallelisation of existing sequential C code, by annotating existing loops with OpenMP pragmas. However, this approach will not produce the best parallel algorithm. A significant rewrite of the code is likely needed.

2.3.3.1 *Data Parallelism*

OpenMP is primarily aimed at data parallelism. Loops are parallelised by annotating them with C pragmas. Instead of providing a set of parallel operations, like TBB or

other skeleton libraries, it provides a skeleton-like parallel for construct, with support for parallel reduction. This is essentially a task farm skeleton. *Modifying clauses* allow the programmer to refine the behaviour of the parallel for, particularly with respect to scheduling and data sharing.

OpenMP's parallel for construct decomposes the iterations of a sequential for loop into parallel tasks. These are then scheduled to the threads of an underlying task pool by the OpenMP runtime. This means that OpenMP hides the boilerplate code needed when using a threading library directly. It can therefore be viewed as a general purpose, low-level, task oriented library that provides lower overhead in terms of code than something like pthreads.

2.3.3.2 *Task Parallelism*

OpenMP allows the programmer to dynamically build unstructured collections of heterogeneous tasks from code fragments scattered arbitrarily throughout the source. This unconstrained model provides no information to assist automatic optimisation, and deviates from the philosophy of parallel skeletons.

2.3.4 *Intel Threading Building Blocks*

Intel's Threading Building Blocks (TBB) (Intel, 2012; Kukanov and Voss, 2007; Contreras and Martonosi, 2008) is a parallel framework that provides a collection of parallel operations, on top of a general purpose task-parallelism framework. It executes a task graph on shared memory multi-core CPUs, using a work-stealing scheduler to distribute work amongst the threads. Parallel operations provided include parallel for, scan, prefix, reduction and pipelines. It provides a C++ template interface for implementing programs. For example, the parallel for pattern takes an input range, and recursively splits it into sub-ranges until a partitioner tells it to stop. It then invokes a functor on each sub-range to generate the final output. Simple instances of this are equivalent to a map skeleton.

TBB uses a work stealing scheduler to allocate tasks to threads and balance computation across cores. Tasks are initially divided equally amongst threads. If a thread completes its allocated tasks, TBB will reallocate tasks from busy threads to the idle thread. This allows TBB to adapt to the underlying machine at runtime, balancing work between threads regardless of how balanced the initial work distribution is.

2.4 PERFORMANCE MEASUREMENT

This section details the techniques and methodologies used throughout this thesis for measuring program performance on multi-program systems. Section 2.4.1 discusses multi-program performance metrics used to compare and contrast different optimisation strategies, and Section 2.4.2 describes the pitfalls of performance measurement on real systems and how these can be overcome. Specifically, methods used to quantify and mitigate noise in the experimental results are described.

2.4.1 *Performance Metrics*

Performance measurement of single program systems is a well established and understood field (Patterson and Hennessy, 1990). Single program performance is usually compared by measuring the execution time of the original unmodified program (T_0) and the execution time of the optimised version of the program (T_1), where execution time is the elapsed real-world time between the start and end of the program's execution.

Performance improvement is then expressed using the *speedup* metric, which is calculated as the ratio of the execution time of the original program to the execution time of the optimised program:

$$\text{speedup} = \frac{T_0}{T_1} \tag{1}$$

This metric produces a single value that can be used to compare optimisation strategies. The optimisation strategy that achieves the highest speedup value is the best strategy.

This metric does not scale to multi-program systems. For example, consider an optimisation strategy that provides a speedup of $2\times$ for program A, and $4\times$ for program B, when the two programs are run concurrently on a shared machine. An alternative optimisation strategy might provide the opposite performance improvement: a speedup of $4\times$ for program A and $2\times$ for program B when run together. Which optimisation strategy is better? This is not clear using individual program speedups. For example, taking the arithmetic mean of the speedups would suggest that the optimisation strategies are equivalently effective. However, what if program A is run more

frequently on the system, or runs for longer? Then clearly the optimisation strategy that provides a larger speedup for program A is the better of the two.

We therefore need a metric that can be used to compare optimisation strategies in the situation where there are multiple concurrently running programs. Average Normalised Turnaround Time (ANTT) and System Throughput (STP) are two system-level metrics that are suitable for examining the performance of multiple concurrently running programs (Eeckhout, 2010). They can be used to compare competing optimisation strategies in such a multi-program environment. They are used throughout this work to compare optimisation strategies.

2.4.1.1 *Average Normalized Turnaround Time*

Average Normalized Turnaround Time (ANTT) is a measure of the perceived slow-down of a set of programs, compared to their individual execution in isolation (Eeckhout, 2010). It is a lower is better metric.

For example, an ANTT value of 1 means that executing the programs in a shared multi-program environment causes no slow-down in the execution time of each program, compared to running the programs in isolation. An ANTT value of 2 indicates that, on average, program execution is slowed down by a factor of $2\times$.

Naively, you may expect an ANTT value of 2 when a pair of programs are run together on a shared system. However, ANTT values between 1 and 2 are possible. For example, one of the programs may not make full use of the computational resources available, due to poor scaling or performing large amounts of disk I/O. A second program can take up this slack, executing useful work while the processor would otherwise be idle.

The ANTT value of a system executing a set of programs P, with execution times T_i^M when run together on a shared multi-program system, and execution times T_i^S when run in isolation on an identical system, is defined as:

$$\text{ANTT} = \frac{1}{|P|} \cdot \sum_{i \in P} \left(\frac{T_i^M}{T_i^S} \right) \tag{2}$$

2.4.1.2 *System Throughput*

System Throughput (STP) is a measure of the rate at which the a system completes work (Eeckhout, 2010). It is a higher-is-better metric.

For example, for a multi-program system executing two programs concurrently, an STP value of 2 means that running the programs in a multi-program environment causes no slow-down to either program. Moreover, the system is capable of completing work twice as quickly, as it completes the execution of twice as many programs in the same amount of time.

The STP value of a system executing a set of programs P, with execution times T_i^M when run together on a shared multi-program system, and execution times T_i^S when run in isolation on an identical system, is defined as:

$$\text{STP} = \sum_{i \in P} \left(\frac{T_i^S}{T_i^M} \right) \tag{3}$$

2.4.1.3 *Average Performance Across a Suite of Benchmarks*

In order to obtain a single performance measurement for a suite of benchmarks, we need to average the ANTT/STP values for each benchmark combination. The harmonic mean of a set of values X is given in Equation 4. This mean is suitable for computing the average ANTT or STP (Eeckhout, 2010), and is used throughout the performance experiments presented in this thesis.

$$\text{HM} = \frac{|X|}{\sum_{x \in X} (1/x)} \tag{4}$$

2.4.2 *Performance Measurement on Real-World Systems*

In this work, experiments are performed on real-world systems. This has the benefit of the demonstrating the optimisation strategies in a realistic setting, however accurate performance measurement in this scenario is difficult. This section discusses why this is, how performance can be measured, and the techniques that can be used to avoid and mitigate the pitfalls of performance measurement on real-world multi-program systems.

2.4.2.1 *Execution Time*

The simplest method for measuring single program performance is to measure the time taken for the program to execute. This can be used to compute the multi-program ANTT and STP metrics discussed in Section 2.4.1. In this work, execution time is measured as the number of seconds that have elapsed between the start and end of a program's execution. Wall clock time is retrieved using the hardware clock provided by modern hardware. These hardware clocks provide a high-resolution time value with at least millisecond accuracy.

2.4.2.2 *Hardware Performance Counters*

In order to gain a deeper understanding of why one optimisation strategy outperforms another, we need more information about the behaviour of the system than solely the execution time of a program. Modern CPU architectures provide hardware counters that can be used to count the frequency of different types of hardware events. For example, the total number of instructions that have been executed, the total number of different types of instruction that have been executed – such as accesses to memory or branch instructions – and cache miss rates at different levels of the CPU's memory hierarchy.

The Performance Application Programmer Interface library (PAPI) (Mucci et al., 1999) provides high-level software mechanisms to interact with hardware counters, and extract their values during program execution with low-overhead. PAPI abstracts away platform specific hardware counter API differences, providing a consistent interface across different architectures. It also supports multi-threading, allowing collection of hardware performance counters for each of a program's threads. This allows us to examine the multi-program scenario that is addressed in this thesis.

2.4.2.3 *Measurement Noise*

Measuring program execution time, or the values of hardware performance counters, across multiple program runs does not result in identical values. This is because real-world multi-program systems are inherently *noisy*. This is caused by a multitude of factors, including the following:

CONTEXT SWITCHING The Operating System scheduler may decide to context switch to another thread at any point in time, and the times at which this occurs is not necessarily the same across repeats. Even if the program is the only user-level

process on the system, the OS scheduler will periodically context switch to run threads from system services.

VARIATION IN INSTRUCTION INTERLEAVING The execution of a parallel program is not necessarily deterministic. Instructions from different threads will not necessarily run with the same interleaving across different repeats, causing variation in execution time.

VARIATION IN MEMORY LATENCY Memory allocation may occur differently across repeats. This will have an impact on caching behaviour and memory access latency, causing variation in execution time across program executions.

INPUT/OUTPUT AND CACHING Programs that access data from permanent storage will vary in execution time depending on whether the data being accessed is cached in memory.

SIMULTANEOUS MULTI-THREADING Many multi-core systems provide multiple hardware threads per core. In this work, this Simultaneous Multi-Threading (SMT) is disabled as it introduces large variability of program execution time between runs.

DYNAMIC FREQUENCY AND VOLTAGE SCALING Modern architectures automatically adjust the clock frequency and voltage levels of the cores, depending on their load, in order to save power without significantly harming performance. This introduces noise, and is therefore disable in this work. The system is configured to run all of its cores at the maximum clock frequency and voltage.

CORE POWER STATES Another technique modern multi-core architectures use to save power is to put unused cores into a low-power or idle state. This introduces noise, and cores in this state take time to switch on when the system workload increases.

2.4.2.4 *Quantifying Measurement Noise*

These properties of the system conspire to mean that the measured performance of the program will differ between runs. Ideally this variation will be small, but in order to derive statistically significant conclusions from the results this noise needs to be quantified. The following sections describe the techniques used in this work to quantify and handle measurement noise in the experiments.

MINIMISING INTERFERENCE The first step is to remove unnecessary system processes from the machine, and ensure no other user processes are executed whilst performance measurements are made. This reduces the frequency and likelihood of the OS scheduler context switching to processes other than those being measured.

REPEATED MEASUREMENTS In order to quantify variation in program performance that is present on real-world, noisy systems, we need to make repeated measurements. This provides us with a *sample* of measurements from which we can use statistical techniques to estimate the real *population* of values.

COEFFICIENT OF VARIATION This is a technique to determine how many repeated measurements to make. The coefficient of variation is a measure of the spread of a probability distribution, relative to the mean.

It is a ratio and is therefore dimensionless, and can be easily used as a threshold for the number of repeats to run without needing to know anything about the population of values.

The coefficient of variation for a distribution is computed using:

$$c_v = \frac{\sigma}{\mu} \tag{5}$$

where σ is the standard deviation of the population and μ is the population mean.

When running real-world experiments, the population standard deviation and mean need to be estimated from the sample that has been collected so far. Therefore, we use the sample standard deviation s and sample mean $\bar{x}$ to provide an estimate for the coefficient of variation:

$$\hat{c_v} = \frac{s}{\bar{x}} \tag{6}$$

A visualisation of the coefficient of variation for two data sets is shown in Figure 5.

In practice, a minimum number of repeats should also be run, so that the sample standard deviation can be computed. In this work, a minimum of 10 repeats are run. Further repeats are then run until the coefficient of variation drops be-

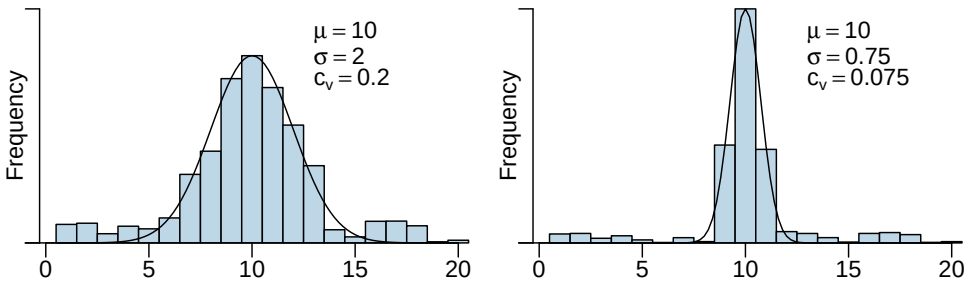

Figure 5: Examples showing the coefficient of variation for two data sets. The blue bars shown the frequencies of values in the data set, and the black lines shows a normal distribution fitted to this data.

low a chosen threshold. For example, using a threshold value of 0.01 means that repeats will be run until the standard deviation drops below 1% of the mean.

CONFIDENCE INTERVALS Confidence intervals are a technique that can be used to quantify the amount of noise in a repeated set of measurements (Georges et al., 2007). For a chosen population parameter (such as the population mean) and sample of measurements, they provide an interval centred on the sample parameter (such as the sample mean) with a given likelihood that the interval lies on the population parameter. This likelihood is controlled by the *significance level*. For example, a confidence interval for the mean with a significance level of 99% is the interval for which there is a 99% chance that the interval lies on the true population mean.

Confidence intervals are used throughout this work to quantify and visualise the errors in performance measurements, such as the ANTT or STP discussed in Section 2.4.1. They are plotted as error bars on relevant plots.

The confidence interval for a sample of values X with parameter θ (such as the mean) and confidence level γ is the interval:

$$[u(X), v(X)] \text{ where } \Pr\left(u(X) < \theta < v(X)\right) = \gamma \tag{7}$$

Under the assumption that the population is normally distributed, the limits of this range are calculated using:

$$u(X) = \bar{x} - \frac{\hat{s}}{\sqrt{|X|}} \cdot \text{PPF}\left(1 - \frac{\gamma}{2}\right) \tag{8}$$

$$v(X) = \bar{x} + \frac{\hat{s}}{\sqrt{|X|}} \cdot \text{PPF}\left(1 - \frac{\gamma}{2}\right) \tag{9}$$

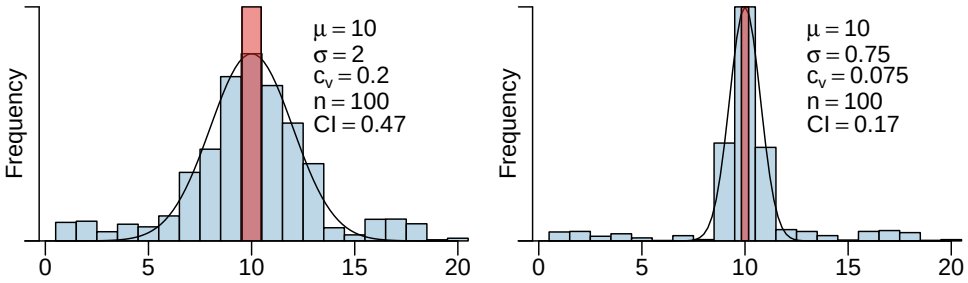

Figure 6: Examples showing 99% confidence intervals for the arithmetic mean of two data sets, shaded in red. The blue bars show the frequencies of values in the data set, and the black lines shows a normal distribution fitted to the data.

where $\bar{x}$ is the mean of sample X, $\hat{s}$ is the standard deviation of sample X. γ is the confidence level – for a 99% confidence interval a value of $\gamma = 0.01$ is used.

A visualisation showing confidence intervals for two different data sets is shown in Figure 6.

For large samples, where the number of measurements exceeds 30, PPF is the probability density function of the standard normal distribution with variance 1 and mean 0. For smaller samples the probability density function of Student's t-distribution with $N - 1$ degrees of freedom is used.

Confidence intervals assume that the noise is normally distributed. This is a safe assumption in this case, as the measurement noise results from the combination of multiple different sources of noise, as discussed in Section 2.4.2.3. Using the central limit theorem, the combination of these noise sources is approximately normally distributed (Rice, 1995).

OUTLIER REMOVAL The mean is often used to provide a single point estimate from the repeated performance measurements. However, it is not a *robust statistic* (Huber, 2005). This means that outliers will have a large effect on its value and cause the range of any confidence intervals to increase in size dramatically. To shrink the confidence intervals to an acceptable size such that conclusions can be drawn from the data would require many more repeats, which may not be practical. An alternative, practical approach is to simply remove these outliers. This is done using *interquartile range removal*. This removes all values that do not lie within the range:

$$[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)] \tag{10}$$

where Q_1 and Q_3 are the values of the 25% and 75% quartiles respectively. A value of $k = 3$ is used in this work.

PERMUTATION OF EXECUTION ORDER As discussed, many repeated program executions are used to quantify measurement errors. However, repeated execution of the same program leads to *systematic errors* in the results.

For example, consider a program that loads data from a hard disk. The first time the program is executed, the input data is read from the disk into main memory, and is automatically cached in the hard disk's cache. On subsequent program executions, the data (or at least part of it) will be read from the buffer cache rather than the disk itself, improving the access latency considerably. This means that the first execution of the program will be significantly slower than subsequent runs.

Systematic errors such as these cause large outliers in the data. They can be turned into random errors – which can be quantified using confidence intervals and the coefficient of variation discussed previously – by randomly permuting the order in which programs are executed.

2.5 SUMMARY

This section has covered the background material necessary to understand the contributions of this thesis, and the relevant background to understand related work, presented in the following chapter.

3

RELATED WORK

This chapter discusses work related to the three cooperative tuning challenges explored in this thesis, and presents a comprehensive review and critical evaluation of prior work in these areas. Section 3.1 discusses existing work on automatically tuning the number of threads used by parallel applications, Section 3.2 explores autotuning of thread placement on multi-socket systems, and in the wider context, and Section 3.3 covers existing work on tuning high-level parameters, and other implementation choices. This chapter concludes with a brief summary in Section 3.4.

3.1 TUNING THREAD COUNT

This section discusses work related to the first of the cooperative tuning challenges explored in this thesis: how many threads should each parallel program utilise? An extensive survey of scheduling techniques is provided by Zhuravlev et al. (2012), and a survey of algorithmic skeleton frameworks, including those which perform thread count tuning, is provided by González-Vélez and Leyton (2010). This section covers the work from these surveys most relevant to our own, and also includes other relevant papers in this area.

In the following discussion, a critical comparison is made between our own thread count tuning approach, which we call ThreadTuner, and existing work. ThreadTuner is an online, adaptive scheduler that chooses how many threads each parallel program running on the system should use, with the aim of maximising overall system performance. Thread counts are chosen based on scalability information for the programs, collected using an offline training phase. The approach dynamically adapts its decisions runtime, in response to changes in the system workload. Runtime overheads are reduced, by using static program information to determine when the system workload changes. ThreadTuner is described in detail in Chapter 4.

Section 3.1.1 begins with a discussion of scalability-based, online approaches. These techniques use program scalability information to decide, at runtime, the number of threads that programs should use. These approaches are able to adapt to changing

system workload. Section 3.1.2 explores thread count tuning approaches that tune the number of program threads statically, using information gathered using offline tuning. These approaches have little runtime overhead as training is performed offline, however they are unable to adapt to changing system workload. Section 3.1.3 discusses approaches which tune a single program, given a fixed system workload. These approaches will adapt a single program to the background workload, however they do not tune the other programs running on the system. The section concludes with a summary discussion in Section 3.1.4.

3.1.1 *Adaptive Scalability-Based Approaches*

Scalability-based approaches tune programs based on information about how program performance changes with thread count. Our thread count auto-tuner, called ThreadTuner, is an example of such an approach.

Sasaki et al. (2012) describe an approach that adjusts the number of threads at runtime using scalability information for each program running on the system. It predicts the scalability of programs, by measuring program scalability at four points in the optimisation space in an offline training phase. The full scalability curve is predicted from these samples using least squares regression. Four sample points are used, as this provides a good trade-off between the length of the training phase and prediction accuracy. The system re-configures the number of cores per program when a program is created or terminated, or a phase change is detected.

Phase changes are detected at runtime by monitoring the cumulative execution cycles per fixed time epoch – referred to as γ in this discussion. Sasaki's phase detection algorithm includes many adjustable parameters including (*i*) the threshold value for changes in γ, (*ii*) the length of each epoch, and (*iii*) the length of an epoch after the thread allocation has changed. These all have an effect on the ability to detect and react to phase changes, and it is not clear how the best values should be chosen. Moreover, γ is a quantity whose absolute values are unknown until a program is executed. It is also likely that the best parameter choices for one system are not portable to other systems.

In contrast, ThreadTuner uses static program information to discover where phases are. This does not require online tracking of program statistics, so does not introduce any runtime overhead. ThreadTuner is also platform agnostic and contains no tunable parameters, as it relies on static program information.

Sasaki et al. (2012) also use a technique called core donation to cope with threads that have been allocated to cores, but which have low utilisation. This allows more than one thread to run on a core, after the initial core allocation has been performed.

ThreadTuner does not do this, instead assuming that a program will make use of its threads as predicted by the scalability information collected in the offline training phase.

Another approach for choosing auto-tuning thread counts is presented by Bhadauria and McKee (2010). Techniques for co-scheduling applications whose performance scales poorly with increasing thread count are investigated. They use performance counters to identify when programs fail to scale, and explore search heuristics to choose which programs to run together, and how many cores/threads to allocate to each program. A fairness metric is proposed, that accounts for the time and space-shared allocation given to a program: some programs may receive occasional allocation of large numbers of cores, while other programs may receive frequent allocation of smaller numbers of cores. The system exploits the fact that some applications experience better scaling while receiving an apparently-unfair resource allocation. Programs to co-schedule are selected based on profiling information from initial sampling runs. The system is evaluated using the PARSEC benchmark suite, however five of the benchmarks are omitted as they scale well. It is argued that evaluating the system's performance using these programs is not necessary, as the best choice is just to gang schedule the programs on all threads. The baseline for comparison in the evaluation is (*i*) running each program with as many threads as there are cores on the system, and (*ii*) the PDPA system developed by (Corbalán et al., 2000), which optimises thread count based on scalability but does not consider other programs running on the system – discussed next.

Corbalán et al. (2000) describe Performance-Driven Processor Allocation (PDPA), a dynamic scheduling strategy for assigning thread counts to programs. Programs measure characteristics about themselves using a tool called SelfAnalyzer. This information is passed to the scheduler, and includes information about (*i*) the number of processors the program would like to be allocated, (*ii*) the speedup achieved by the current processor allocation, and (*iii*) an estimate of the total execution time of the program. The PDPA scheduling policy is applied periodically, once every fixed time epoch. This limits the speed with which their approach can react to changes in system workload, unlike our approach which is based on static information from the structure of programs.

Pusukuri et al. (2013) present a scheduling framework, called ADAPT, for co-scheduling multi-threaded applications on multi-core machines. ADAPT uses supervised learning techniques to predict the effects of interference between programs, and for differences between applications' own scaling due to lock contention. ADAPT produces models for programs' performance based on their behaviour with sample input-output values, aiming to predict the time spent productively in user-mode – rather than in the kernel, for example in synchronisation functions. These models are used to determine how many threads a set of parallel workloads should use. It continuously monitors programs and changes the thread allocation in response to changes in system workload. This contrasts to our approach which determines when system workload changes based on static program information.

In summary, the work by Sasaki et al. (2012) is most similar to our own. They use a similar scalability-based method, however use a runtime-based approach to detect program phases, which differs to our static approach.

3.1.2 *Static Approaches using Offline Training*

This section covers static approaches for choosing the thread count of parallel programs. These approaches are incapable of reacting to change in system workload, as they make tuning decisions before programs are executed.

Moore and Childers (2012) aim to optimise system throughput, whilst meeting user-specified quality of service (QoS) constraints, which are intended to avoid issues such as starvation and to provide more resources for programs with more demanding QoS requirements. Their approach first builds a utility model for every program running on the system, using an offline training phase. Each program is run with varying thread counts, with a synthetic benchmark providing competing workload on the system.

This synthetic benchmark approach is not necessarily representative of the actual workload that will run on the system. Their approach requires an expensive training phase, using program profiling. Their approach is not input agnostic, and depends heavily on the programs used to train and build the utility model. The system is evaluated using a small subset of the PARSEC benchmarks: blackscholes, bodytrack, canneal, streamcluster and swaptions. The other benchmarks are not used due to limitations in available inputs and thread configuration flexibility.

The approach is compared to three alternative strategies: (*i*) *free-for-all*, where each program spawns as many threads as there are cores on the system, (*ii*) *uniform partition* where each program gets an equal share of the available cores, and (*iii*) *per-application maximum* where each program is given as many cores as possible without harming performance when executing in isolation on a system. For example, if a program scales well up to 8 cores, and then drops in performance, it is given 8 cores. Note that this doesn't take into account other programs running on the system.

Lee et al. (2010) present ThreadTailor, which tunes the thread counts for programs on invocation. When a program starts, its thread count is optimised for the current system workload. It is then fixed for the execution of the rest of the program. Their approach requires offline training to predict the type of threads and the communication patterns that they will use at runtime. It also assumes that the best choice for the program at invocation time is the best choice throughout the execution of the program. However, as system workload on these systems varies, this is unlikely to be the case. This is in contrast to our approach, which tunes programs whenever a change in the system workload occurs. Their evaluation only uses and small, and eclectic, collection of programs for evaluation: three programs from the PARSEC benchmark suite, two benchmarks from related work and one from the SPLASH2 benchmark suite.

Wang and O'Boyle (2010) also describe a static tuning approach using machine learning. It tunes both the number of threads and scheduling strategy.

ThreadTuner improves over these approaches in its ability to react to changing system workload, which is commonplace on shared-memory multi-core systems. Offline trained approaches are unable to react to these dynamic changes, and therefore unable to provide optimal performance across applications running on the system. One argument in favour of offline trained approaches is that the runtime overhead they incur is minimal. However, the overhead introduced by ThreadTuner is negligible, and this cost if offset by the performance improvements that its adaptability provides.

3.1.3 *Fixed System Workload*

This section discusses approaches that tune a single program, and treat the other programs running on the system as a fixed workload. This contrasts strongly with our approaches, which modify all programs running on the system in order to obtain

the best overall performance. However, assuming that the background workload is fixed is useful for legacy systems, where existing applications cannot be modified.

Emani and O'Boyle (2013) use a machine learning model, using static program and runtime features, to optimise a target program given a fixed workload of programs that are already running on the system. The existing workload programs do not adapt their resource allocation, only the target program. It's not clear how their approach adversely affects the workload already on the system. Their approach is not platform agnostic, as it requires re-training on new systems. Their evaluation uses the NAS parallel benchmarks and the C benchmarks from SPEC OMP 2006, all of which are implemented using OpenMP. Leave-one-out cross-validation is used to train and test the machine learning model. The evaluation only reports speedup for each program. It does not use a system-wide metric, such as ANTT or STP. Three types of workload are explored: light, medium and heavy, and two arrival patterns: high frequency and low frequency. The baseline for the evaluation is the default Linux OS scheduler, with the number of threads spawned equal to the number of cores on the system. They also compare against a simple hill-climbing approach to optimise the target program It is claimed that the system beats the technique proposed by Raman et al. (2012) as they do not need to explore different thread configurations at runtime. Instead, they are statically predicted and used to inform the runtime decisions using compiler knowledge.

Grewe et al. (2011) explore adaptation of the number of threads to system workload for OpenMP applications. A machine learning based model is used to predict the number of threads for a given application and system workload. The model is based on a set of static program features and dynamic workload features. The workload features simply consist of the total number of threads and the total number of programs executing on the system. They target two objectives, and build different models for each. The first maximises program performance regardless of workload, and the second maximises program performance without adversely affecting the execution of the external workload. The largest cost associated with their approach is training the machine learning model. As set of benchmarks are used to build the model, by executing them with different numbers of threads and different workloads. It is not clear if the set of program features they use includes high-level behaviour and properties of the algorithm. However, it could be argued that this is sufficient as the approach is targeting OpenMP, which is a framework for task-based parallelism. Better performance may yet be possible if high-level properties of the program are

considered, and more complex transformations (other than just modifying the number of threads) are considered. Again, this approach only adapts to the workload present when the application is first invoked. Better performance may be attainable by adaptively modifying the program to changes in workload throughout the execution of the program.

All of these approaches make a serious assumption: that the system workload is fixed. This is not true for multi-program shared-memory systems, where the system workload is constantly evolving. These approaches could be extended to cope with changing workload, although it is not clear how effective they would be. For example, the approach by Emani and O'Boyle (2013) could be concurrently applied for every program in the system. However, the decision made for each program may not be the best decision for the system as a whole, as the scheduling decision is made independently for each program under the assumption that the decisions made for other programs are fixed.

3.1.4 *Discussion*

Related work in the field of scalability-based thread count tuning discussed in Section 3.1.1 is the most similar to our own. These works use program scalability information to decide the schedule at runtime, allowing them to adapt to changing system workload. In particular, (Sasaki et al., 2012) present a scalability-based tuning approach similar to our own, however use a different mechanism for the detection of program phase changes. Their approach detects phases at runtime, and includes several tunable parameters that affect phase detection accuracy. In contrast, our approach uses static program information to predict where phase changes occur, and includes no parameters.

Alternative approaches consist of static tuning approaches (Moore and Childers, 2012; Lee et al., 2010) but these are not capable of adapting to the changing system workload present on multi-program multi-core systems.

There is also work targeting legacy systems where the system workload cannot be changed (Emani and O'Boyle, 2013; Emani, 2014; Grewe et al., 2011), however these approaches are not useful in the context of systems where all of the running programs have a malleable thread count.

3.2 TUNING THREAD PLACEMENT

The previous section discusses work related to choosing how many threads each parallel program should use. This section discusses work related to the second of the cooperative tuning challenges explored in this thesis: where should each program's threads be executed? Zhuravlev et al. (2012) provide an extensive survey of work in this area. Here we highlight the work most relevant to our own, and additional recent papers. In the following discussion, a critical comparison is made between our own spatial scheduling approach, which we call LIRA, and existing work. LIRA is presented in detail in Chapter 5.

This section begins in Section 3.2.1 with a discussion of Callisto, a parallel runtime that aims to reduce interference between programs. LIRA is built on top of Callisto. Section 3.2.2 describes demand balancing approaches, which aim to equalise some performance metric across the system. Our LIRA scheduler is an example of such an approach, as it aims to balance the rate at which programs execute memory load instructions across processor sockets. Section 3.2.3 discusses observation-based scheduling, where the performance achieved by different thread to core mappings are directly measured to find the best choice, either at runtime or during an offline training phase. The remaining sections discuss thread placement tuning in the wider context. Section 3.2.5 explores hardware based and analytical techniques, and Section 3.2.4 discusses cooperative scheduling in the context of virtual machines. Although these areas are not directly related to our own work, they are an important area of the cooperative scheduling field and are included for completeness. This section concludes with a summary discussion in Section 3.2.6.

3.2.1 *Callisto*

Harris et al. (2014) describe Callisto, a parallel runtime that aims to reduce the interference between program threads running on a shared multi-core system with a single socket. Callisto's thread scheduler treats the system as a homogeneous array of cores and arbitrarily assigns programs to cores. Callisto reduces scheduler-related interference by reducing lock-holder pre-emption problems, by reducing load imbalance between worker threads within a program, and by making explicit thread-to-core allocations which adapt to the amount of parallelism available within a program. This approach helps the system achieve good utilization in the presence of bursty par-

allel workloads, and demonstrates that most scheduler-related interference between pairs of workloads on a shared-memory multi-core machine can be avoided.

Callisto is not applicable to multi-socket systems, as it does not consider the non-uniform nature of communication between cores in the system. LIRA uses Callisto as a baseline for building a socket-aware thread to core scheduler, described in detail in Chapter 5. Callisto's thread scheduler is described in more detail in Section 5.4.1, and the performance issues it encounters on multi-socket systems are discussed in Section 5.4.2.

3.2.2 *Demand-balancing Approaches*

Demand-balancing approaches to cooperative scheduling aim to equalise a chosen performance indicator across the entire system. For example many techniques, including LIRA, aim to balance a measure of memory system pressure across sockets.

The scheduling approach by Bhadauria and McKee (2010) discussed in more detail in Section 3.1.1, is also relevant here. They track hardware performance counters to decide how to place thread on cores at runtime. In contrast to our approach, they combine scalability information rather than focusing solely on memory pressure.

Zhuravlev et al. (2010) investigate scheduler-based techniques for addressing interference between threads sharing a common last level cache. They observe that, in addition to contention for space in a cache, contention at memory controllers, memory buses, and prefetching hardware could also be significant. This observation led them to focus on a thread's solo last level miss rate to predict the extent to which it will suffer from contention, since a higher miss rate will tend to stress the downstream parts of the memory system. They examine scheduling policies which sort threads by miss rate, and distribute them such that the total miss rate is equal across last level caches. In addition, they describe an online scheduling algorithm which dynamically measure miss rates, rather than using statistics from solo runs. LIRA uses a similar performance metric as a predictor of the pressure that a program will place on the memory system. Like LIRA, their online scheduler, called DIO, is capable of responding to changes in system workload, by detecting changes in a programs miss rate. Although these approaches are similar, they are applied in different contexts. DIO considers scheduling threads on a multi-core processor, whereas LIRA is focused on scheduling for multiple sockets, balancing the performance indicator across sockets rather than cores.

Banikazemi et al. (2008) describe a system to monitor CPU performance counters to identify patterns which indicate poor behaviour. Under various assumptions, they estimate the cache occupancy ratios of different threads based on information available from CPU performance counters, from which they estimate the likely impact of moving a thread between cores. These moves are made experimentally, reverting to a default scheduling policy if an anticipated improvement is not seen. This approach differs greatly from our own. Banikazemi et al. (2008) actively probe the schedule to see if performance is positively or negatively affected. In contrast, LIRA attempts to accurately predict what the best program to core mapping is without the overhead involved in probing for the best allocation strategy. However if LIRA makes a misprediction it is likely to cause bad system performance as it will not detect and react to its mistake.

Knauerhase et al. (2008) co-schedule 'light' and 'heavy' programs on a processor where pairs of cores share a last level cache. Their experiments suggest that the number of cache misses per cycle is the best indication of interference, and they aim to equalise this metric across the cores. As with Fedorova et al. (2007), they increase the amount of CPU time given to 'light' threads which were co-scheduled with 'heavy' tasks, reflecting the fact that 'light' tasks were more likely to suffer from interference. Again, this approach is similar to our own in that it attempts to balance a chosen performance metric across the programs being scheduled. However, it is not explained how programs are categorised as either 'light' or 'heavy'. LIRA does not require programs to be categorised - it operates dynamically at runtime, without the need for any training overheads.

McGregor et al. (2005) examine the problem of selecting pairs of threads to place together on a simultaneous multi-threaded processor. They examine bus transactions per thread, stall cycles per thread and last level cache miss rates per thread. Their system uses a scheduler to select which sets of programs to run in a given quantum, attempting to balance the chosen metric between quanta. Within a quantum, a spatial scheduler places a thread with a high value for the metric with another thread with a low value. Their results suggest that focusing on stall cycles was effective, perhaps reflecting the fact that contention between the pairs of hyperthreads the most significant factor. Our work is similar in approach, in that we aim to balance a performance indicator, namely the rate at which load instructions are executed, across programs. However, our approach is focused on the allocation of programs to cores, unlike Mc-

Gregor et al. (2005) who focus on improving performance of threads running on a single hyperthreaded core.

Dey et al. (2013) describe ReSense, a system for dynamically controlling the number of threads used by concurrent applications. Their sensitivity to sharing memory resources is characterised from single-program run-alone measurements designed to stress each resource. Thread-to-core mappings are controlled when applications start or stop, or when threads are created or destroyed. In contrast to LIRA, their approach requires an expensive offline training phase.

Libutti et al. (2014) explore a user-mode resource management mechanism to select which workloads to co-schedule. The machine is divided into a series of binding domains (BDs) comprising memory and cores, with BDs either allocated to individual processes, or shared between them. Applications are characterized by application working modes, and an optimisation algorithm used to select which application to operate at which mode. Unlike LIRA, an offline training phase is needed. Their evaluation focuses on just two benchmarks from the PARSEC benchmark suite.

Corbalán et al. (2001) adjust the number of threads used by OpenMP programs, in order to improve gang scheduling. Their approach performs spacial scheduling within each time slot of the gang schedule. Their evaluation only considers 4 programs taken from SPECfp95 and the NAS parallel benchmark suite.

Tian et al. (2009) study the use of A* search to explore possible schedules for sets of programs.

In summary, there are many competing approaches, including our own, that use a measure of memory system pressure to influence thread placement tuning. Many of these approaches require an offline tuning phase. This requires an accurate training set to be devised, to avoid issues such as over-fitting. Our approach is a heuristic approach that bases its decisions on information collected at runtime, without the need for training.

Cache Partitioning and Memory Management

The following approaches also perform demand-balancing, but adjust the cache partitioning and memory management of the system, rather than scheduling of programs and threads. This is a different approach to LIRA, which only alters the thread to core mapping to improve memory usage patterns.

Xie and Loh (2008) introduce a taxonomy of applications: (*i*) *turtles* which do not make much use of a shared last level cache, (*ii*) *sheep* which can exhibit a high rate

of last level cache accesses, but which achieve a low miss rate when allocated a small number of ways, (*iii*) *rabbits* which are very sensitive to the number of ways allocated to them, and degrade in the presence of contention, and (*iv*) *devils* which access the cache frequently and have high miss rates. They use this classification in a cache partitioning algorithm to improve cache usage patterns. The aim of the cache partitioning is to identify *devils* and contain them within a partition of the cache. This isolates their poor cache locality from other programs running on the system. This is a radically different approach to our own, which instead attempts to balance cache usage across cores without restricting programs to a subset of the cache. It also focuses on cache usage within a single shared cache, rather than across multiple-sockets as in our work.

Lin et al. (2008) provide another approach to classify programs and perform cache partitioning, based on the performance degradation programs observe when running with a reduced size level 2 cache. 'Red' and 'yellow' programs are harmed by reduced cache space. 'Green' programs are not slowed down so much, but have a miss rate of at least 14 misses per thousand cycles. 'Black' programs have low slow down and low miss rate. Page colouring is used to control the amount of cache space allocated to different applications. Their work provides insights into the reasons for particular performance interactions, as it is based on hardware runs rather than simulation of components in isolation. Their approach requires an expensive offline tuning phase, in contrast to LIRA which uses performance information collected at runtime.

Dashti et al. (2013) present Carrefour, a memory management algorithm which considers the placement of memory within non-uniform memory access (NUMA) systems, and the conflicting goals of providing low latency access by placing memory close to threads that will access it, versus spreading memory across a machine to achieve higher aggregate bandwidth. They consider pairs of programs running on a single machine, but programs workloads run on complete NUMA sockets. This is a different approach to improving performance, which focuses on placement of memory, rather than placement of threads. This work is orthogonal to LIRA, and it would be interesting to combine their techniques with our own.

These approaches, whilst similar to LIRA, differ in that they are used to decide cache partitioning strategies rather than the placement of threads. They are therefore incapable of tuning where threads are placed on a multi-socket system and so do not mitigate the issues surrounding multi-socket systems, such as reducing inter-socket communication overheads.

Tuning for Data Centres and Clusters

The following techniques perform demand-balancing scheduling in large scale data-centres. This context is somewhat different to the context in which LIRA is evaluated, namely a single multi-socket multi-core system.

Tang et al. (2011) examine the impact of sharing memory in data-center applications. They observe larger performance differences between co-scheduling decisions in these workloads than in PARSEC benchmarks. They examine many reasons for these properties, and include heuristics to predict good thread-to-core mappings from characteristics gathered when running alone. Latency-sensitive applications are placed first, based on the resources which they seem to benefit from, for example if an applications requires high bus usage. Batch jobs are then placed, aiming to use different resources. Unlike LIRA, their approach does not adapt to changing system workload as, once the scheduling decision has been made, it is fixed.

Mars et al. (2010) also investigate cooperative scheduling on clusters. They address contention between mixes of latency-sensitive and batch programs. They record the last level cache miss rate for the latency sensitive program and, if it is high, the batch program is de-scheduled. Various algorithms are explored, including sampling the latency-sensitive program performance during an interval where the batch program is temporarily prevented from running. This technique allows the latency-sensitive program to be profiled in effect in isolation, without requiring dedicated characterisation runs. However their evaluation only considers quad-core CPUs, which may not be representative of the types of systems used in cluster-based computing. The background workload used in their evaluation is the same across benchmarks. A more thorough evaluation is needed to judge whether their approach works in general.

The work by Mars et al. (2010) is similar to our approach but uses a different performance criteria to control the scheduling: namely program latency. However this approach assumes that overall system performance will be harmed when a latency sensitive program is co-scheduled with another program. It also completely de-schedules one of the programs, which is likely to lead to a large performance hit in terms of ANTT. Our LIRA approach instead aims to directly predict whether a pair of applications will perform well when run on the same socket, and does not necessarily take the drastic step of completely penalising one application. LIRA therefore attempts to improve both ANTT and STP, whereas Mars et al. (2010) aim to improve solely STP.

Frachtenberg et al. (2005) investigate scheduling MPI applications running on networked clusters of machines. They observe that applications that communicate a lot with one another should be scheduled to run concurrently on the same nodes, otherwise communication can be delayed due to one of the applications not running. On the other hand, applications with little communication can be more flexibly scheduled to cope with load imbalance on the cluster. This system does not perform runtime adaptation as, for example, the number of nodes used by each application is fixed.

3.2.3 *Observation-based Co-scheduling*

This section discusses observation-based co-scheduling, where the performance achieved by different thread to core mappings are directly measured to find the best choice, either at runtime or during an offline training phase. This is in contrast to our approach, which predicts the schedule that will achieve the best performance, without needing to directly measure the performance, either at runtime or in an offline training phase.

Klug et al. (2011) describe Auto-pin, a tool for adaptively pinning threads to cores to obtain the best performance. Auto-pin maintains a candidate set of *pinnings*, or mappings from program threads to cores. The initially large space of pinnings is pruned using architectural information. For example, all symmetrical pinnings are ignored as it is assumed they provide equivalent performance. Auto-pin then measures the performance of each candidate pinning. It does this by dividing the executing into fixed time epochs, and using a different pinning for each. It then chooses the best pinning and uses it for the remainder of the program's execution.

Auto-pin therefore adapts a program to the system workload on invocation. It doesn't modify the pinning after the best pinning has been chosen, and so does not continuously respond to changes in workload at runtime. This tool can also be used when multiple programs are concurrently executing. Newly invoked programs will tune themselves such that they perform well alongside another auto-pinned program.

However, the existing auto-pinned program will not adapt its pinning to the presence of the new program. Furthermore, when the existing auto-pinned program terminates the pinning for the new program is unlikely to be optimal. It is unclear how auto-pin will perform when two or more programs are exploring candidate pinnings concurrently.

As they discuss in the conclusions of the paper, the approach will suffer with applications that go through distinct phases of execution. By applying a runtime adaptation technique to skeletons, this is avoidable as the skeleton dictates the computation and communication patterns, and will therefore dictate any phase changes. They suggest running a re-pinning if performance drops below a certain threshold. This may help alleviate this problem, however it is unclear how effective this would be.

Snavely and Tullsen (2000) explore co-scheduling of threads in simultaneous multi-threaded systems. Their system dynamically explores different combinations of thread placement. This approach can be used on unmodified hardware, and without the scheduler needing to understand the causes for the performance results seen. However, unlike LIRA, a profiling phase is required. Also, unlike LIRA, it does not adapt to changing system workload – which is essential for achieving optimal system performance in multi-program systems.

Bulpin and Pratt (2005) describe an algorithm for co-scheduling on simultaneous multi-threaded processors which collects performance-counter and timing information from pairs of benchmarks running together on a single processor with two hardware threads. They show how this information can be used to dynamically select which pairs of benchmarks to run together, assuming more software threads than hardware threads. This small scale study only consider single core, dual threaded processors, unlike LIRA which is applied in the context of multi-core multi-socket systems.

3.2.4 *Virtualisation*

Virtualisation is another field where cooperative scheduling is important. This is quite a different environment to the multi-socket environment in which LIRA operates, however it is still useful to explore approaches in this area as they utilise many of the same concepts and develop similar co-scheduling ideas.

Dhiman et al. (2009) describe vGreen, a system that places virtual machines within a cluster of physical machines running a virtual machine monitor. They observe that co-scheduling heterogeneous virtual machines is effective both from the viewpoint of performance and energy consumption. They classify virtual machines based on the number of memory accesses per cycle. Live migration is used to adjust placement decisions dynamically.

Merkel et al. (2010) introduce *task activity vectors* to characterise the resource requirements of computationally-intensive single-processor virtual machines. In their evaluation they examine memory bus, L2 cache, and 'un-shared' core resources taken collectively. The vector is maintained dynamically based on performance counter values. Threads are spread to balance the demands at different positions in the vector. In addition to migrating virtual machines between nodes, they consider the choice of clock frequency on the different vector components and on the energy delay product of the workloads as they run. This balancing approach is similar to the approach taken by LIRA, and other demand-balancing based approaches described in Section 3.2.3, however it is applied in the context of single processor virtual machines.

Nathuji et al. (2010) describe Q-Clouds, which identifies interference between co-scheduled virtual machines based on application-level indications of their performance. Workloads are characterised in isolation on a staging server to determine which resources are needed for which levels of performance. Techniques such as page colouring are used to isolate workloads, and a 'head room' of unused capacity is left mitigate other sources of interference.

Zhang et al. (2013) use a cycles-per-instruction (CPI) metric to identify 'bad' applications running on machines in a shared compute cluster. In their setting, hundreds of instances of a process are run across a cluster, and so it is possible to use statistical approaches to identify performance problems if one process' CPI differs from that of other processes in the same job.

3.2.5 *Analytical and Hardware Approaches*

The following techniques perform co-scheduling using hardware based and analytical techniques. These are not directly related to our work, but are an important aspect of the field of cooperative scheduling, and so are covered here for completeness.

Suh et al. (2002) describe an early mechanism for combining time-shared and space-shared scheduling in multi-processor systems. They propose additional hardware monitoring to estimate the miss-rates as a function of cache size, both for processes running along and in combination using cache-partitioning techniques. Qureshi and Patt (2006) propose hardware to dynamically monitor applications to let software predict the impact of different cache-partitioning algorithms. LIRA does not require any custom hardware, as it utilises hardware counters available on most modern CPUs.

Chandra et al. (2005) show how to combine stack-distance or circular-sequence profiles from pairs of threads to predict the number of cache misses that will occur when pairs of threads run together. This provides an analytical basis to anticipate interference, but it requires a mechanism to collect profiling information, and does not account for interference aside from cache misses. In contrast, our use of load instruction rates as a proxy for memory pressure provides a metric that captures more about the memory system than just cache misses.

Jiang et al. (2008) prove that, under various assumptions, a general form of the optimal co-scheduling problem is NP-complete, while a special case restricted to pairwise-interactions is possible in polynomial time.

3.2.6 *Discussion*

Related work in the field of observation based co-scheduling, discussed in Section 3.2.2 are the most similar to our own. However, none of these works use load instruction rate as a proxy for memory pressure. This simple, low-overhead runtime metric is an effective way of balancing memory pressure across sockets, it can be measured with negligible overhead, and does not require custom hardware as it is available on many modern CPUs.

Lin et al. (2008) explores cache partitioning techniques, using a similar strategy to LIRA but applied to memory system rather than thread scheduling. Dashti et al. (2013) also investigate cooperatively sharing resources, by adjusting memory placement. Tang et al. (2011); Mars et al. (2010) explore approaches for data-centres and cluster computing, with a focus on solely improving the performance of latency sensitive applications rather than overall system performance.

In outline, we use the observations of work such as that of McGregor et al. (2005); Knauerhase et al. (2008); Lin et al. (2008); Banikazemi et al. (2008); Zhuravlev et al. (2010) that jobs with "heavy" demands for shared resources should not be placed close to one another. We exploit this observation, aiming to balance memory system pressure across sockets, using an online adaptive approach that does not require a profiling step, as is the case for some the approaches by Knauerhase et al. (2008); Dey et al. (2013); Libutti et al. (2014).

3.3 TUNING HIGH-LEVEL PARALLEL PARAMETERS

The previous section discusses work related to choosing where to schedule threads for parallel programs running on a shared system. This section discusses work related to the third cooperative tuning challenge explored in this thesis: tuning of high-level parallel framework parameters. Relevant work from the extensive surveys by González-Vélez and Leyton (2010); Zhuravlev et al. (2012) is covered, along with additional recent papers. Section 3.3.1 explores related works on auto-tuning high-level parameters. Section 3.3.2 explores auto-tuning in the wider context of adjusting the implementation of parallel algorithms.

3.3.1 *Tuning High-Level Parameters*

The following approaches tune high-level parameters in parallel frameworks. These are the approaches most similar to our own.

3.3.1.1 *FastFlow*

FastFlow by Aldinucci et al. (2013) is a stream parallel processing framework for multi-core shared memory machines. Its main contribution is the use of lock-free fence-free queues to pass tasks between threads. It is most suitable when fine grained parallelism is present. The queues introduced very minimal overhead into the computation, allowing the algorithm to be decomposed into small tasks.

It is implemented as a library consisting of multiple layers of abstraction. At the lowest level, it provides a memory allocated specifically suited to the allocation of many small tasks. Higher layers provide a threading model (built on POSIX threads) and the lock-free fence-free queue implementation that can be used for communication. At the highest layer, a collection of high-level skeletons are provided.

FastFlow contains several implementation parameters that are manually tuned, and are amenable to auto-tuning (Collins et al., 2012).

3.3.2 *Tuning the Implementation*

The following frameworks perform modification of the implementation of parallel programs, using a range of techniques. They include compilers that generate platform specific codes to runtime approaches that dynamically adapt the implementation.

3.3.2.1 *Delite: Parallel Domain Specific Languages*

Delite (Chafi et al., 2011; Brown et al., 2011) is a parallel programming framework which allows the creation of domain specific parallel languages (DSLs), embedded in Scala (Rompf and Odersky, 2010). The authors identify three, often conflicting, goals in program language design: generality, productivity and performance. Delite deliberately trades off generality to provide productivity and performance. However, it allows the generality that it trades off to be modified depending on the use case. This is done through support for the creation of arbitrary DSLs.

Delite provides a core set of parallel operations, which can be extended to implement high-level domain-specific parallel operations. Delite includes low-level optimisations that can be applied to the core set of parallel operations, and the ability to define domain-specific optimisations for user-defined domain-specific operations. These are expressed as pattern matching rules. This allows cooperation between expert programmers for a given architecture, and domain experts who are designing a DSL.

A Delite program achieves platform independence by leaving the compilation until walk-time (Fisher, 1997). A program includes source code for different target architectures, which are then compiled when the program is invoked depending on the exact hardware available.

Delite expresses all of its operations in terms of a single operator: the MultiLoop. This pattern iterates over a range, applying a function to each index in the range. It then performs an optional reduction step over thread-local data. It can read from multiple inputs and produce multiple outputs. It is key in allowing some optimisations, such as fusing of parallel operations.

3.3.2.2 *PetaBricks: Programmer-Provided Algorithmic Choice*

PetaBricks (Ansel et al., 2009) is a language, compiler and runtime for array computation that allows the programmer to provide a choice of implementations for their algorithm to the compiler and runtime system. Additional tuning parameters can also be manually specified. For example, consider a sorting algorithm. This removes the need for a programmer to insert hand-coded cutoffs between different sort implementations of their algorithm: the best approach is to use quicksort for large arrays and a simpler sorting algorithm for small arrays. PetaBricks can find these cut-offs automatically, if the programmer manually exposes this tunable parameter.

PetaBricks performs its tuning at compile time, using micro-benchmarking to tune the parameters. This allows it to adapt to different inputs at runtime, however it cannot adapt to changing system workload. It is also targeted at single program optimisation.

3.3.2.3 *SkePU: Automatic Implementation Variant Selection*

Dastgeer et al. (2011, 2013) present a tuning approach for choosing between skeleton implementations. Their framework provides a set of commonly used skeletons, each of which is implemented in four variants: sequential C++, OpenCL, OpenMP and CUDA. The framework chooses an appropriate variant at compilation time. Their approach performs tuning at compile time, and so cannot respond to changes in system workload. It also ignores the presence of multi-programs.

3.3.2.4 *Copperhead: Synchronisation and Shape Analysis*

Copperhead (Catanzaro et al., 2011) is a data-parallel skeleton language, consisting of a heavily restricted subset of Python and a set of data-parallel primitives. Copperhead's main contributions are static optimisations called *synchronisation analysis* and *shape analysis*. Copperhead performs these optimisations statically. It does not perform runtime optimisation, and so does cannot cope with changing system workload. These optimisations are also applied in a single-program context, unlike our approach.

3.3.2.5 *Thrust: Templates for CPU and GPU*

Thrust Nvidia (2012) is a parallel algorithms library, providing an analogue of the C++ Standard Template Library, that aims to be a productive yet high performance language. It aims to provide performance portability across different devices, including CPUs and GPUs.

Thrust allows the programmer to choose whether to execute a parallel operation on either a CPU or GPU. However, this choice is down to the programmer and is hard coded into the application. Thrust is therefore unable to choose based on system workload.

3.3.2.6 *Adaptive Execution for SMT Processors*

Jung et al. (2005) develop a technique to adaptively alter loop parallelisation techniques using performance monitoring. Their approach identifies the best execution

strategy at runtime, when the program is invoked, to determine the best execution strategy for the loops in the program. Delite goes a bit further by making this choice at *walk-time*. This compiles the program for a given architecture when it is invoked, but does not modify it at runtime. This approach also does not make its decisions based on dynamic properties of the system workload, only static properties such as the hardware that is available.

3.3.2.7 *Summary*

In summary, related work in this area only adapts the implementation of parallel programs in a single-program context. The majority of work is also restricted to performing this tuning at compile time, except the work by Jung et al. (2005) which tunes loop parallelisation at runtime.

3.3.3 *Discussion*

Related work in the area of high level parameter tuning, discussed in Section 3.3.1 are the most similar to our own. These approaches vary implementation parameters at runtime in order to improve performance. Other related work in this area is focused on switching the implementation used by the skeletons in order to accommodate the underlying hardware (Nvidia, 2012; Dastgeer et al., 2011, 2013) or generating code for the given system Jung et al. (2005). These optimisations are applied at compile time, or at walk-time in the case of Delite Chafi et al. (2011); Brown et al. (2011). This contrasts with our observation that, in a shared multi-program environment, this tuning needs to be performed at runtime in order to deal with changing system workload.

3.4 SUMMARY

This chapter has provided a critical review of related work in the areas explored in this thesis. The next chapter presents the contributions of this thesis for the first of the three challenges: cooperatively tuning the number of threads used by programs.

4

COOPERATIVELY TUNING THREAD COUNT

This chapter, the first of three technical chapters, presents work to address the first cooperative auto-tuning challenge: how to choose the best number of threads for each parallel program running on a shared-memory multi-core system.

A scheduling framework, called ThreadTuner, is developed. It improves system performance by intelligently choosing how many threads – and therefore cores – each program running on the system should use. The ANTT and STP metrics are used to measure performance, and are described in detail in Section 2.4.1. ThreadTuner is an online adaptive scheduler, that responds dynamically to changes in the system workload at runtime. Two implementations of this auto-tuning technique are implemented. The first for our own multi-threaded C++ skeleton framework, called ThreadTuner-Skel, and the second for the libgomp implementation of OpenMP, called ThreadTuner-OMP.

The rest of this chapter is structured as follows. Section 4.2 motivates this work, by demonstrating the performance gains available from system-wide tuning of program thread counts. Four techniques for assigning thread counts to programs are evaluated for a set of image processing benchmarks. Section 4.3 investigates the main challenge for choosing the number of threads for parallel applications: their performance scalability as the number of threads is adjusted and its sensitivity to program phase changes. Section 4.4 describes an auto-tuning technique for choosing the best number of threads for a set of programs, given information about the scalability of program phases. Program phases are determined statically from the structure of the skeleton parallel programs. Section 4.5 describes our multi-threaded C++ skeleton library that allows dynamic reconfiguration of the number of threads used by programs, and implements the ThreadTuner tuning method. Section 4.6 presents an implementation of ThreadTuner for the libgomp implementation of OpenMP (Dagum and Menon, 1998), which we call ThreadTuner-OMP. Section 4.7 presents an evaluation of both implementations of our auto-tuner. The performance achieved by the framework is compared to the default OpenMP scheduler. ThreadTuner-Skel is evaluated using a set of image processing benchmarks, which consists of skeleton implementations of algorithms from OpenCV Bradski and Kaehler (2008). ThreadTuner-OMP is evalu-

ated using the NAS parallel benchmark suite Bailey et al. (1991). Section 4.8 concludes the chapter with some closing remarks.

4.1 INTRODUCTION

This chapter explores the first of the three cooperative tuning challenges addressed in this thesis: how many threads should each parallel program use, when running on a shared-memory multi-core system?

The choice of program thread counts is vital to achieve optimal system performance on shared-memory multi-core systems, as it affects the following performance issues: (*i*) under-subscription, where insufficient threads exist on the system to exploit the available hardware resources, (*ii*) over-subscription, where too many threads exist on the system causing conflicts for resources, and overhead in thread management and inter-thread communication, and (*iii*) thread scalability, where increasing the thread count for a program results in diminishing returns in performance, or in some cases decreased performance.

Existing approaches either choose the number of threads used by multi-threaded parallel programs under the assumption that they are running exclusively on the system, or the choice of thread count is made individually by programs with no coordination with the system as a whole. For example, to maximise program throughput, a program will likely spawn a number of threads equal to the number of cores available on the system. However, this can lead to poor performance when multiple parallel programs that use this approach are running on the system. Many more threads than cores will be present on the system, causing over-subscription and degradation of performance due to issues including synchronisation, cache usage patterns and context switching overheads. Typical OS schedulers schedule the threads that they are given, and do not have direct control over how many threads programs will use.

What is needed is a framework that chooses the best number of threads for each program running on the system, such that overall system performance is improved. This decision needs to take the number of cores available on the system and thread scalability of the running programs into account. Moreover, this tuning needs to be performed dynamically, due to the dynamic nature of the system workload present on shared-memory multi-core systems (discussed in more detail in Section 1.1) and the phased behaviour of many parallel programs.

This chapter presents ThreadTuner, an auto-tuning framework that addresses these issues by intelligently choosing the number of threads used by each program running on the system. It aims to improve overall system performance, by minimising the ANTT of the system.

ThreadTuner auto-tunes the thread counts of parallel programs based on their *thread scalability*. This is the variation in program performance as the number of threads it uses is changed. Programs scale well if their performance increases as the number of threads used is increased. Scalability information if used to choose the number of threads used be each program currently executing, to ensure that the available system resources are fully utilised effectively and without over-subscription. For example, programs with poor scalability are assigned fewer threads than those with better scalability, but in a way that avoids starving the poorly scaling program of all computational resources. Our auto-tuning framework will only allocate as many threads as there are cores on the system, therefore avoiding the problems caused by over-subscription.

The scalability of program *phases* within programs are also considered. Phases in the execution of the programs running on the system are identified, and the tuner updates its decision every time one of the programs starts execution, completes execution or transitions between phases.

ThreadTuner is applied to skeleton parallel programs. These are implemented using a structured approach to parallel programming that separates algorithm description from implementation, described in more detail in Section 2.3. These structured parallel programs allow ThreadTuner to predict statically where phase changes occur in programs, making the tuning problem tractable. The sequential code between skeletons also constitute a different phases of execution. These code regions are used for program initialisation and to glue together different skeleton instances. Section 4.3 provides an empirical investigation into the thread scalability of the phases in a suite of parallel programs.

Information about the scalability of program phases is collected a priori, in an offline training phase. This scalability and phase information is used by the scheduler to choose how many threads each parallel program should use during a given phase of execution. This is done dynamically at runtime, and so can adapt to the changes in system workload present on shared-memory multi-core systems. The thread allocation decision is reviewed whenever a program starts execution, completes execution or experiences a phase change.

This framework is implemented for both a multi-threaded C++ parallel skeleton library, called ThreadTuner-Skel, and OpenMP, called ThreadTuner-OMP. Across 16 randomly chosen pairs of programs from the NAS parallel benchmark suite (Bailey et al., 1991) run on a 12-core shared-memory multiple core system, ThreadTuner-OMP achieves an ANTT of 2.63 and an STP of 1.03. This is compared to the libgomp OpenMP implementation using the default Linux OS scheduler, which achieves a ANTT of 2.91 and STP of 0.98 across the same program combinations. ThreadTuner-OMP provides an improvement in ANTT of 11% compared to libgomp OpenMP, and a marginal improvement in STP.

4.2 SCHEDULING STRATEGIES

Consider a multi-core system on which we want to run a pair of parallel programs (called A and B) which are malleable – i.e. their thread counts can be dynamically changed at runtime. A takes longer to execute than B when run in isolation on the system utilising all of the available cores.

The standard approach taken by many application developers is to spawn as many threads as there are cores on the system, and have the OS scheduler interleave their execution. This helps utilise all of the systems resources, however this over-subscription can be harmful if many programs are running concurrently.

Consider the case where programs A and B are run concurrently on the system. Each program spawns N_A and N_B threads respectively. The threads are pinned to distinct cores to avoid performance effects caused by the migration of threads between cores, and the total number of threads does not exceed the number of cores on the system – to avoid issues of over-subscription. When program B completes its execution, a phase change occurs in the system workload. The resources that were being used by program B are now free for program A to use. It is likely beneficial for A to spawn more threads in order to utilise all of the cores on the system, increasing its thread count to N'_A.

In this example, we need to choose the optimal values for the thread counts N_A, N'_A and N_B. For now, we choose these parameters by doing an exhaustive search of the space, and later develop a technique to predict these values, presented in Section 4.4. The ANTT and STP system performance metrics can be used to determine whether one choice of thread counts performs better than another.

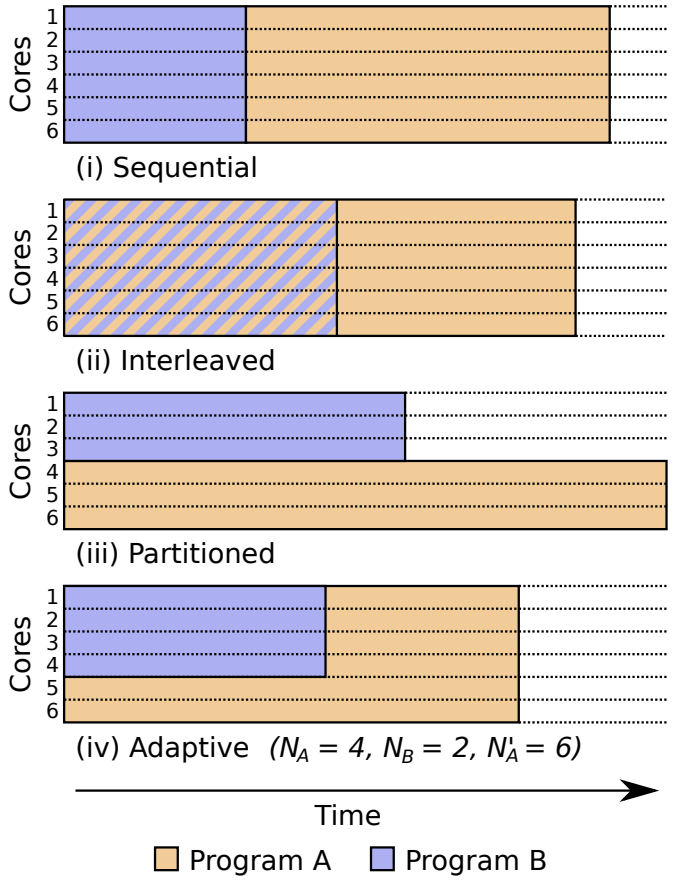

Figure 7: Four scheduling strategies for allocating programs to cores. Each colour shows the execution of a different parallel pattern instance. (*i*) simply runs one program followed by the other, allocating them the all available cores, (*ii*) interleaves the execution of the threads in time using the Operating System's scheduler, (*iii*) partitions the cores into two, allocating half of the cores to each program and (*iv*) initially partitions the cores into two subsets (thread counts N_B and N_B, but changes the allocation when the first program completes execution (thread count N'_A).

To motivate the need for thread count tuning, we invesigate the following scheduling strategies for choosing the thread counts N_A, N'_A and N_B. These scheduling strategies are depicted in Figure 7.

INTERLEAVED Each program spawns as many threads as there are cores, and pins its threads to distinct cores. Therefore two threads are pinned to each core. The programs run concurrently, and the execution of the pair of threads on each core is interleaved in time by context switching between them at regular intervals, as controlled by the OS scheduler.

SEQUENTIAL The threads are partitioned in time. A is run to completion, utilising as many threads as there are cores in the system, followed by B which also spawns as many threads as there are cores.

PARTITIONED A number of threads equal to the total number of cores on the system is used by programs A and B. These threads are pinned to distinct cores. The programs run concurrently, using a subset of the systems resources. For these experiments, each program is allocated an equal number of threads, so they use a equal share of the available resources.

ADAPTIVE Initially, A and B spawn a number of threads (N_A and N_B) and pin them to distinct cores. When B completes execution, program A spawns additional threads to utilise the now idle cores, with thread count N'_A. The choice for the parameters N_A, N_B and N'_A is determined by an exhaustive search of the space of possible values. The set of thread count parameters that provide the minimum total execution time are chosen.

We evaluate these scheduling strategies using combinations of programs, chosen based on their thread scalability characteristics. Figure 8 shows the scalability of a set of image processing benchmarks. It demonstrates that different applications scale differently to the number of cores. These use algorithms taken from the OpenCV image processing library (Bradski and Kaehler, 2008), and implemented using the multi-threaded C++ skeleton library described in Section 4.5.

Each of these program pairs combines one program with good scalability and another that has poor scalability. The program with good scalability will benefit from using as many resources as possible, whereas the amount of resource allocated to the poorly scaling program will need to be carefully tuned. This should expose the flaws in the default OS scheduler as it will not take this information into account.

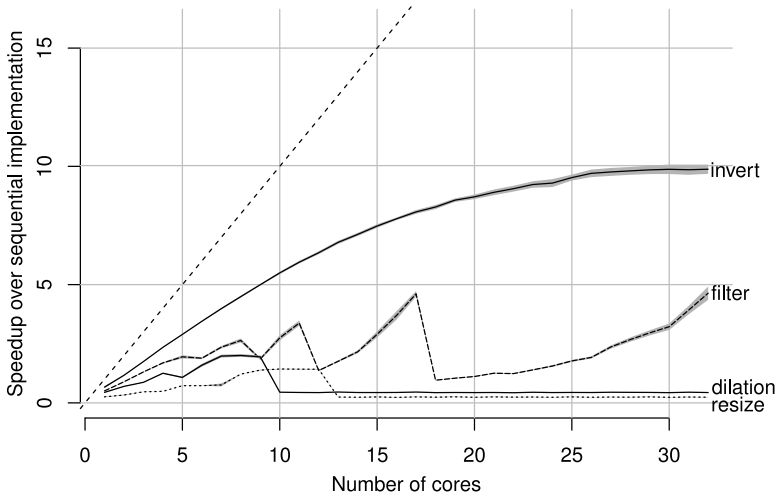

Figure 8: Scalability of a variety of benchmarks on a 32-core shared memory machine, compared against a sequential implementation. The dotted line shows perfect linear scaling.

From this set of image processing benchmarks, we use EROSION, INVERT, FILTER and RESIZE. The EROSION and RESIZE benchmarks exhibit poor scalability – as the number of threads increases, performance also increases, until a sweet spot is reached after which point increasing the number of threads harms performance. The FILTER benchmark also exhibits poor scaling. Its scalability curve has many peaks and troughs. The INVERT benchmark exhibits good scaling – as the number of threads increases, the performance increases but with diminishing returns as the thread count gets larger.

Figure 9 shows the performance achieved by each scheduling strategy. The experiments were run on a 32-core shared-memory multi-core machine. As expected, the simple PARTITIONED strategy, which allocates a static partition of the cores to each program, achieves a much significantly worse system throughput than the other approaches. This is expected as this scheduling approach does not make use of the resources that become available after one of the programs completes execution. However, it does provide a similar ANTT to the adaptive approach.

ADAPTIVE performs best of all of the strategies, when considering both ANTT and STP. On average it achieves an ANTT of 2.64 and an STP of 0.62, compared to the Linux OS scheduler (INTERLEAVED) which achieves a significantly higher ANTT of 2.98, and a lower STP of 0.55. It achieves comparable ANTT to the partitioned approach, but significantly better STP.

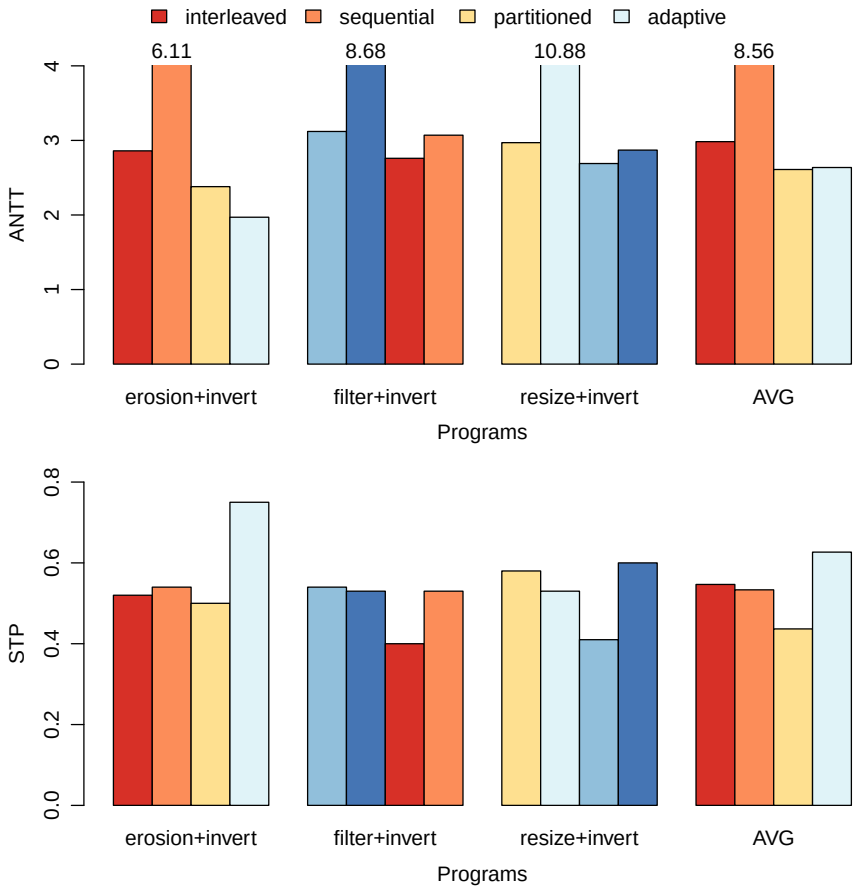

Figure 9: Performance achieved by each scheduling strategy, showing the ANTT and STP achieved by each, and the arithmetic mean across program combinations. For ANTT lower is better, and for STP higher is better. The experiment was run on a 32-core shared memory multi-core machine. Each program combination is run for multiple repeats until the coefficient of variation drops below 0.01. Error bars show 99% confidence intervals for this mean.

In all three cases, ADAPTIVE achieves better ANTT than INTERLEAVED. In two of the three cases, ADAPTIVE achieves better STP than INTERLEAVED. For FILTER+INVERT it performs slightly worse, however it does improve ANTT in this case.

The SEQUENTIAL scheduling technique achieves poor ANTT in all cases. This is due to the fact that one of the programs must wait for the other program to execute before its execution begins. This strategy does not provide responsiveness. However, it provides reasonable STP results, comparable to the INTERLEAVED strategy (the Linux OS scheduler).

These motivational experiments demonstrate the need for cooperative auto-tuning of thread counts for parallel programs. The best system performance is only achieved when considering the thread allocation for all of the programs in a cooperative manner. Moreover, an adaptive approach is needed to react in changes to system workload.

4.3 THREAD SCALABILITY

Parallel program performance is affected by the number the number of threads that the program can utilise. This section provides an empirical exploration of the scalability of skeleton parallel programs, and investigates the presence of different phases of execution and the differing scalability of these individual phases. This analysis backs up the assumptions in the design of the THREADTUNER scalability-based tuning approach described in Section 4.4, and extends the observations made in Figure 8 to the more established NAS parallel benchmark suite (Bailey et al., 1991).

4.3.1 *Scalability of Whole Programs*

Figure 10 shows the scalability of programs from the NAS parallel benchmark suite. This data was collected on a 12-core shared-memory system. To quantify noise in the measurements, each program is run for multiple repeats, until the coefficient of variation drops below 0.01. See Section 2.4.2.4 for details.

This plot demonstrates that different programs scale differently. The EP benchmark scales well, suffering only a slight slow-down even for large thread counts. In contrast FT scales poorly. Increasing the number of threads for this benchmark initially provides good scaling, but beyond 6 threads the performance increase obtained by increasing the thread count becomes vanishingly small. Also of note is SP, which ini-

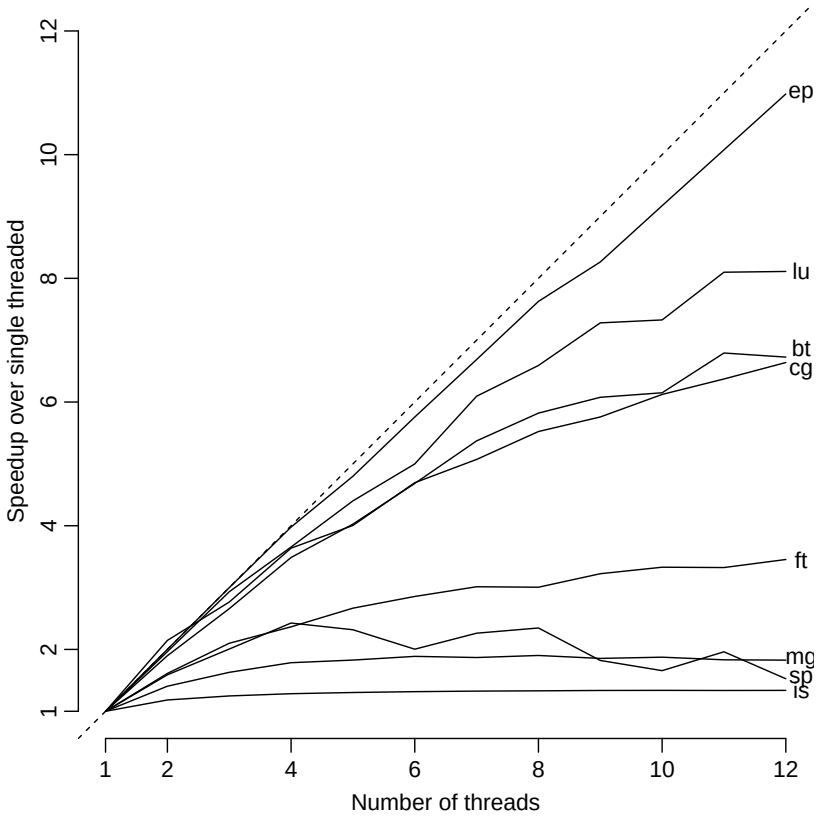

Figure 10: The thread scalability of the NAS parallel benchmarks, showing how different programs exhibit different performance trends as the number of threads is varied. The plot shows speedup over using a single threaded against the number of threads. Programs were run in isolation on a 12-core shared-memory multi-core system. Each sample was run for multiple repeats, until the coefficient of variation dropped below 0.01. The solid lines show the mean execution time, and shaded regions show the 99% confidence interval for this mean.

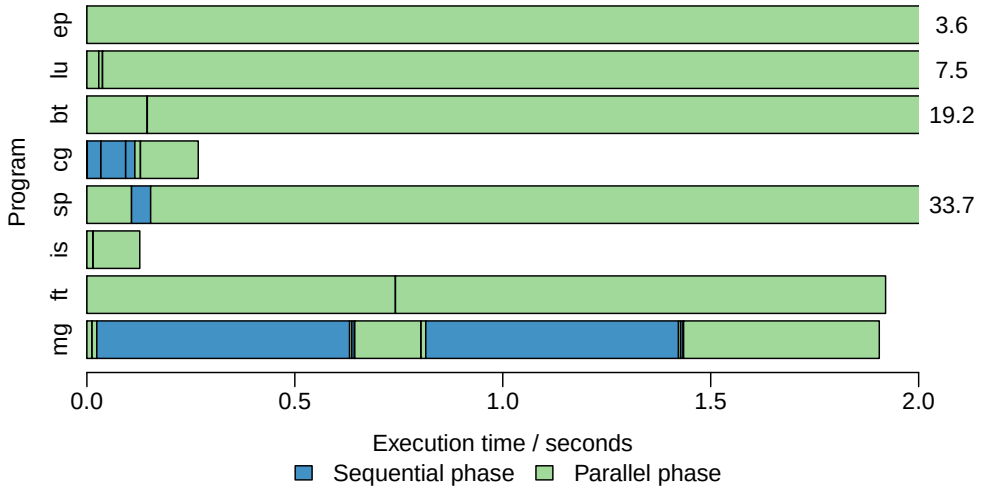

Figure 11: Sequential and parallel execution phases within the NAS parallel benchmarks. The plot shows whether each program is in a sequential or parallel phase at different points throughout its execution. Each phase is bordered by a black box.

tially scales up to 4 threads. Increasing the thread count further results in a drop in performance.

These results demonstrate that scalability information needs to be taken into account when deciding how many threads each program should be allocated. For example, if EP and SP are to be run on the same system, a naïve approach may allocate an equal subset of the system to each program, providing 6 cores to each. However, due to the poor scalability of SP beyond 4 threads, this will not achieve optimal performance. A better choice would be to assign 4 cores to SP and the remaining cores to EP.

4.3.2 *Scalability of Program Phases*

Figure 11 shows a visualisation of the phases within the benchmarks explored in the previous section. This plot demonstrates that each program has multiple parallel phases, and exhibits a variety of phases. Some have long sequential regions, and some have several parallel regions. For example, EP exhibits a single parallel phase, preceeded by an almost negligible sequential set up phase. In contrast, MG consists of several long-lived sequential and parallel phases. This motivates the need for regular thread allocation decisions given the frequency with which the applications change phases. This plot also shows that most phases take a significant amount of time to

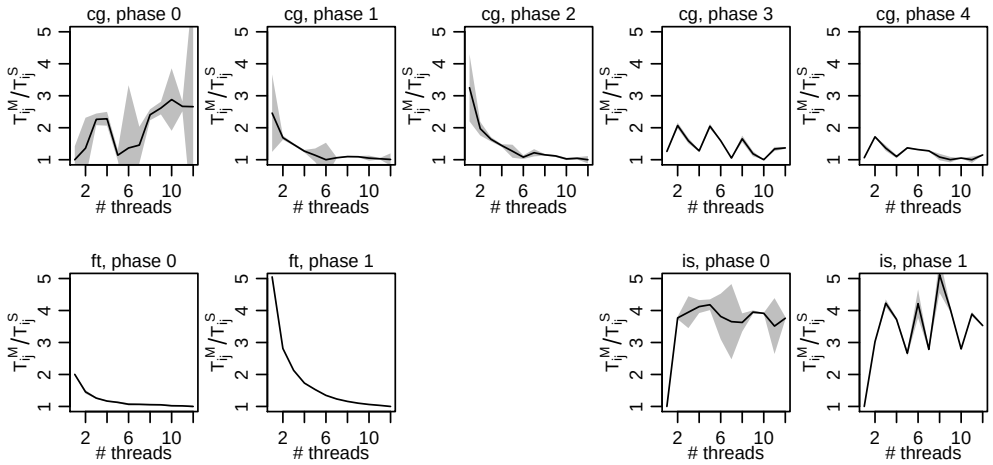

Figure 12: Thread scalability for the separate parallel execution phases in the NAS parallel benchmarks (*i*) CG (top row), (*ii*) FT (bottom left) and (*iii*) IS (bottom right). Each plot shows the number of threads used against the turnaround time for each parallel phase of each benchmark. The phases are in program order, and sequential phases are omitted. Turnaround time ($T^M_{i,j}/T^S_{i,j}$) is the term summed when computing ANTT, and is the term that must be minimised to minimise the ANTT. Each sample is run for multiple repeats, until the coefficient of variation is below 0.01. The solid line shows the mean turnaround time, and shaded grey regions show 99% confidence intervals for this mean.

execute. Amdahl's law therefore suggests that it is worthwhile to carefully tune the performance of these phases in order to achieve performance gains.

Figure 12 shows the scalability of the phases from a subset of the NAS parallel benchmarks. These plots demonstrate that different phases within parallel applications exhibit different scalability. This motivates the need to carefully choose the thread counts for parallel programs not based on entire program scalability, but on the scalability of the program phases that are currently executing on the system.

4.4 SCALABILITY-BASED COOPERATIVE AUTO-TUNER

The previous section demonstrates that different programs exhibit different thread scalability, i.e. the increase in performance obtained by increasing the number of threads differs between programs. Moreover, it is demonstrated that programs consist of several phases, each with different thread scalability. This section presents a cooperative auto-tuning technique that exploits this phase scalability to improve overall system performance, specifically the ANTT metric discussed in Section 2.4.1.

4.4.1 *Cooperatively Auto-Tuning ANTT*

The tuning problem that needs to be solved is as follows. Given a system where multiple programs are running, choose a number of threads for each program that minimise the ANTT of the system. Moreover, this decision needs to be made every time a program exhibits a phase change in its behaviour. Detection of these phase changes is discussed in Section 4.5.2.

The formula for computing the ANTT of the system, and a detailed discussion of this system-wide performance metric, is given in Section 2.4.1 Equation 2. This formula is duplicated below:

$$\text{ANTT} = \frac{1}{|P|} \cdot \sum_{i \in P} \left(\frac{T_i^M}{T_i^S} \right)$$

Here T_i^S is the best execution time of program i when running exclusively on the system (a single-program environment), and T_i^M is the execution time of program i when running alongside other programs on the system (a multi-program environment).

In order to optimise overall system performance, we need to choose thread counts for each running program that minimise the system's ANTT. Given the equation for ANTT, this is equivalent to minimising the sum of terms T_i^M.

T_i^S is fixed for each program, whereas T_i^M varies depending on the number of threads allocated to each program on the system. T_i^S therefore does not play a part in the choice of thread allocation. We can instead discover the values of these terms via direct measurement of T_i^S a priori or by predicting it at runtime. The number of programs $|P|$ is also fixed in this equation.

Minimising ANTT is therefore equivalent to minimising the terms T_i^M which are the execution time of each program i when run together on the system. We therefore need to determine the relationship between the number of threads allocated to a program and the value of T_i^M, which will allow us to choose the thread allocation that minimises ANTT.

However, measuring this relationship a priori is impractical as the space is, for all practical purposes, infinite. This is because any combination of programs could be running on the system at the same time. This makes the problem intractable, however it can be simplified by constraining the system such that each program is allocated a subset of the available cores in the system. This modification to the space of possible thread allocations provides a way to predict the relationship using

single-program execution data collected a priori. By assuming that programs do not interfere with one another, the terms T_i^M can be predicted from the execution time of programs run in isolation. We therefore only need to determine the relationship between the execution time of each program and the number of threads allocated to it, independently for each program that is to be run on the system.

Given this scalability information for each program we can compute an estimate for the ANTT for every possible thread allocation. Moreover, we can do this at runtime and therefore choose the optimal thread allocation for the system to use. We simply use the scalability information to compute the ANTT for every possible thread allocation, and pick the thread allocation that minimises ANTT. This is the thread allocation that we use for the current programs executing on the system.

4.4.2 *Tuning Program Phases*

This scalability based tuning technique can be extended to handle program phases. The execution time of a program i is simply the sum of the execution time of its phases Q_i:

$$T_i^S = \sum_{j \in Q_i} \left(T_{i,j}^S \right)$$

and similarly for execution in a multi-program environment:

$$T_i^M = \sum_{j \in Q_i} \left(T_{i,j}^M \right)$$

The ANTT for the system can therefore be expressed as:

$$\text{ANTT} = \frac{1}{|P|} \cdot \sum_{i \in P} \left(\sum_{j \in Q} \frac{T_{i,j}^M}{T_{i,j}^S} \right)$$

As for the whole-program case, minimising this equation is equivalent to minimising the terms $T_{i,j}^M$, as P, Q_i and $T_{i,j}^S$ are invariant in the equation.

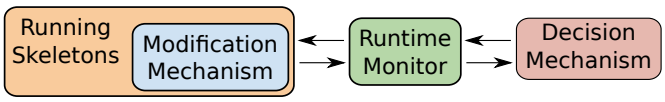

Figure 13: Diagram showing the components of the system and their interaction.

4.5 SKELETON-BASED IMPLEMENTATION

This section describes the implementation of our thread count tuner, called THREADTUNER-SKEL, for our skeleton library. The skeleton library is a multi-threaded C++ library for image processing. Figure 13 outlines the components in the tuning system. The runtime monitor informs the decision mechanism about phase changes in the running programs, and therefore prompts it to make resource allocation decisions. The modification mechanism provides an interface for the decision mechanism to modify the resource allocation of the running programs. The design of the modification mechanism is described in Section 4.5.1, the runtime monitor is covered in Section 4.5.2 and the decision mechanism in Section 4.5.3. The decision mechanism implements the cooperative thread tuning algorithm presented in Section 4.4.

4.5.1 *Modification Mechanism*

The modification mechanism provides an API via which the thread count of a running application can be modified.

Figure 14 shows how the skeleton modification mechanism controls the number of cores utilised by a skeleton instance. The work to be executed is represented as a range of tasks. For example, this could be pixels from an image being processed by a stencil skeleton. Each core is initially allocated a range of these tasks to execute. For example each core could be given separate parts of an input image to process. Each core executes its tasks in order, shrinking the range of remaining tasks. When the modification mechanism changes the number of cores, it re-allocates these ranges of tasks so that they are divided amongst the new set of available cores. This may leave unavailable cores on the system that can be utilised by other applications.

4.5.2 *Runtime Monitor*

The purpose of the runtime monitor is to communicate *events* and *performance metrics* to the decision mechanism. It also relays decisions from the decision mechanism to

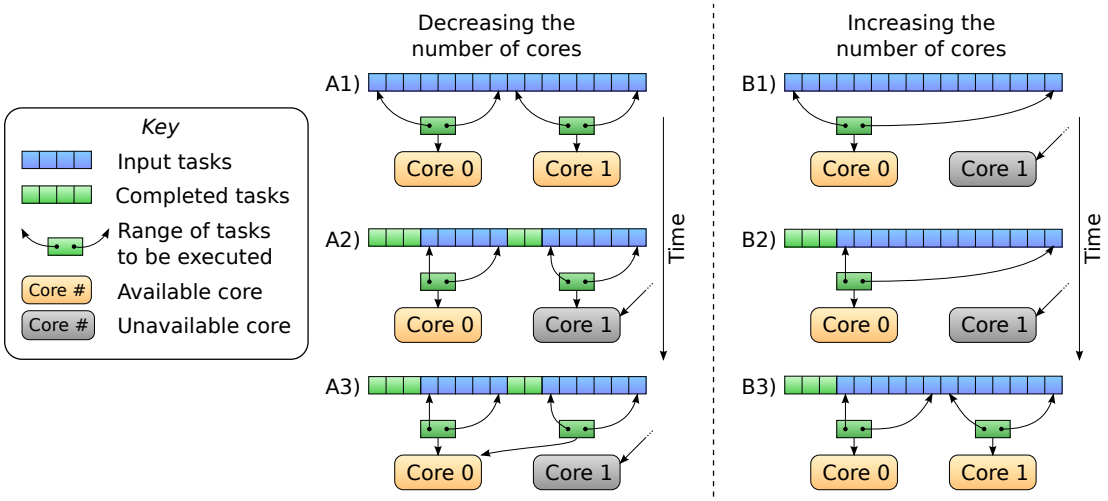

Figure 14: Diagram show how the modification mechanism adjusts the number of threads used by running skeleton instances. A1) and B1) show the initial thread allocations. A2) and B2) show the situation after some of the threads have completed tasks. A3) demonstrates what happens when the number of threads allocated to a program is decreased, and B3) demonstrates what happens when the number of threads allocated to a program is increased.

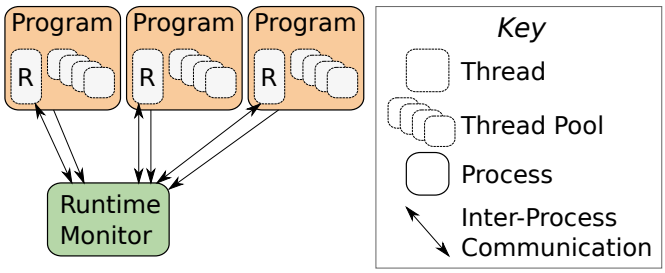

Figure 15: Structure and communication between running programs and the runtime monitor component. Programs communicate phase change events and performance metrics to the runtime monitor, and the runtime monitor sends thread allocation decisions to the programs.

programs, so that they can modify their implementation at runtime to meet their new resource allocation.

The communication structure between each program and the runtime monitor is detailed in Figure 15. There are two channels of communication between each program and the runtime monitor. The main program process sends *events* to the runtime monitor (detailed below). A separate reconfiguration thread marked 'R' in the diagram listens for reconfiguration requests with a new resource allocation from the runtime monitor, and responds with an acknowledgement when the skeleton has been reconfigured.

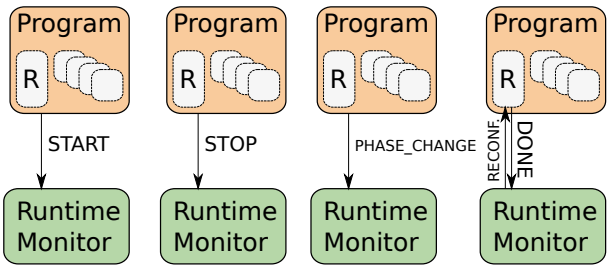

Figure 16: Diagram showing the messages communicated between the runtime monitor and currently executing programs.

Events consist of the following:

- A new skeleton starts.
- A skeleton completes execution.
- A skeleton has a phase change in its behaviour.

The rationale behind these is that the resource allocation for the system only needs to change when there is a change in the workload of the system. This is the case when either a new skeleton starts executing (increasing the demand on the resources of the system), and skeleton finishes execution (decreasing the demand on the resources of the system) or a skeleton has a phase change in its behaviour.

Phase changes are programmed statically in the skeleton implementation. For example, consider the prefix skeleton applied to a 2-dimensional volume. It first computes partial sums across the rows of the volume, and then computes the output across these columns of partial sums. This skeleton has two phases: computation of the rows, and computation of the columns. Therefore, a phase change event is sent to the runtime monitor between these two phases. This is hard coded in the prefix skeleton implementation by the library developer.

The communication performed between the runtime monitor and running programs is summarised in Figure 16. Each program has a thread that listens for reconfiguration messages from the runtime monitor, marked 'R' in Figure 16. When one occurs, it uses inter-thread communication to tell the thread pool to stop, reschedule its work according to the new resource allocation, and then continue processing work. Once the thread pool is reconfigured, thread 'R' informs the runtime monitor that the reconfiguration has completed.

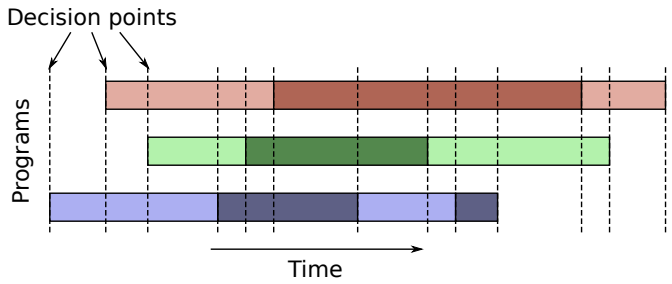

Figure 17: Visualisation showing the points during program execution at which the decision mechanism updates its allocation decision. Each program is coloured differently, with light and dark regions indicating execution phases within the parallel program that exhibit different thread scalability.

4.5.3 *Decision Mechanism*

The first issue to address is to decide when the decision mechanism should make resource allocation decisions. We assume that the execution of each program consists of a sequence of phases. Each phase has different behaviour and characteristics, meaning that the best resource allocation for each phase differs. However, within a single phase, we assume that the behaviour is static, and therefore the optimal resource allocation remains fixed.

These observations simplify the tuning problem, as we are only required to make a decision about resource allocation whenever a program changes phase. This occurs when a program starts execution, changes parallel pattern or completes execution. This is visualized in Figure 17. Note that because our programs are implemented using parallel skeletons, we statically know where the phase changes are. This avoids having to do complex and expensive phase detection at runtime.

4.5.3.1 *Handling Sequential Phases*

A further consideration that needs to be addressed is the presence of sequential program phases. Programs often contain sequential phases, such as initialisation code at the start of the program or between parallel phases. Our tuning approach handles this by keeping track of whether a phase is parallel or sequential. If it is sequential, the thread allocation is fixed to a value of 1. If it is parallel, then our tuning approach is free to choose the optimal thread allocation.

4.5.3.2 *Making Tuning Decisions*

Given these properties of the ANTT metric, the auto-tuner needs to choose a thread count for each currently running phase such that the sum of terms $T_{i,j}^{M}$ is minimised.

To make the tuning problem tractable, we can allocate a distinct subset of the system's cores to each currently executing program phase, and assume that the execution of each program phase is independent. This allows us to estimate the multi-program turnaround time for each phase ($T_{i,j}^{M}$) using the single-program turnaround time of each phase for a given subset of threads (t_i) measured a priori ($T_{i,j}^{S}(t_i)$). This scalability information can then be used to choose the best thread count. The best choice are the thread counts (t_i) that minimise the ANTT of the system, predicted using the single-program turnaround times collected a priori:

$$\text{Predicted ANTT} = \frac{1}{|P|} \cdot \sum_{i \in P} \left(\sum_{j \in Q} \frac{T_{i,j}^{S}(t_i)}{T_{i,j}^{S}} \right)$$

4.5.3.3 *Tuning Algorithm*

The decision mechanism chooses the number of threads for each currently running program as follows.

The decision mechanism runs as a separate process on the system, and sets up a pipe for communication with running programs. When a program starts, it informs the decision mechanism by sending it an appropriate message over the pipe. Also, when a program experiences a phase change, it sends a phase change message to the decision mechanism over the pipe.

When the decision mechanism receives a message, it chooses a new thread count for every running program, such that the predicted average normalized turnaround time of the system is minimised. This is calculated using the formula given in Section 4.5.3.2. This is done by estimating the average normalized turnaround time of the system for every possible allocation of thread counts to the currently running programs. The thread counts that provide the minimum estimated ANTT are chosen as the new schedule. Messages are sent to each running program, using the pipe, informing them of how many threads they are allowed to use.

For programs that are in a sequential phase, they are trivially assigned a thread count of 1. Programs in a parallel phase are assigned at least one thread, such that the total number of threads used by all programs does not exceed the number of cores in the system.

4.6 OPENMP IMPLEMENTATION

This section describes the libgomp OpenMP implementation of our dynamic auto-tuner, called ThreadTuner-OMP. Due to the lack of mature benchmarks for our multi-threaded C++ skeleton programming library, we also implemented the tuning technique for OpenMP so that it can be evaluated the mature NAS parallel benchmark suite (Bailey et al., 1991). While OpenMP does not provide as much high level program information and tuneable parameters as a skeleton parallel program, it is sufficient to demonstrate our approach. It has one runtime adjustable parameter (number of threads) and provides high level phase information (OpenMP parallel regions). We implemented our adaptive tuning system as part of GCCs OpenMP runtime library libgomp and evaluated it using the mature NAS parallel benchmark suite.

The runtime monitor and modification mechanism, whose implementation was described in the previous section, have been implemented as part of libgomp. The runtime monitor communicates phase changes to the decision mechanism using inter-process communication. It does this whenever an OpenMP parallel region starts or stops. The modification mechanism listens for thread allocations from the decision mechanism, and suspends or resumes threads as required.

All of the inter-process communication is implemented using the high-level libzmq inter-process communication library (Hintjens, 2013). This is a lightweight communication library that provides a range of communication patterns, such as publisher-subscriber and point-to-point.

Our implementation within OpenMP has one important constraint. Parallel loops must use the dynamic scheduling strategy. This allows the modification mechanism to check for new resource allocations from the decision mechanism at regular intervals during execution of the parallel loop. However, it will incur some execution overhead when a program is running exclusively on the system, as work cannot be statically allocated to the threads in the program.

The decision mechanism itself is implemented as a Python script that listens for phase change event messages from the running programs on the system. When a phase change message is received, the script uses the previously collected scalability information to choose the new optimal thread allocation. It then communicates the new thread allocation using IPC back to the running programs. The phases within programs are determined by the decision mechanism, which is part of the libgomp

library. It sends a phase change message to the scheduler script whenever a top-level parallel for block is entered, or completes execution.

4.7 EVALUATION

In this section, we analyse the performance of both implementations of our scalability-based tuning technique: ThreadTuner-Skel which auto-tunes our parallel skeleton library and ThreadTuner-OMP which auto-tunes OpenMP applications. Section 4.7.3 investigates the overhead incurred by the inter-process communication between running programs and the runtime monitor, for ThreadTuner-Skel. Section 4.7.4 examines the effect of changing the frequency of inter-process communication on this overhead, for ThreadTuner-Skel. Section 4.7.2 evaluates the ANTT and STP achieved by both ThreadTuner-Skel and ThreadTuner-OMP compared to the default Linux OS scheduler.

4.7.1 *Experimental Setup*

For our experiments we use a single-socket machine, with a 12 core Intel E5-2620 processor clocked at 2GHz. The system has 16GB of main memory and runs Linux 3.7.10. We use GCC 4.7.2 to compile the benchmark programs.

We use benchmark programs taken from two different sources. Firstly, we use a suite of image processing benchmarks implemented using our multithreaded C++ skeleton framework. This consists of algorithms taken from OpenCV (Bradski and Kaehler, 2008). The second set of benchmarks are taken from the NAS parallel benchmark suite, and are used to evaluate the OpenMP implementation of the auto-tuner discussed in Section 4.6.

To quantify the error in our measurements, we run each program for at least 3 repeats, possibly more, until the coefficient of variation falls below 0.01. This technique is described in more detail in Section 2.4.2.4. We measure the time for the entire execution of each program, including loading data from disk.

4.7.2 *NAS Parallel Benchmarks*

Figure 18 shows the ANTT and STP achieved by our adaptive tuning system. A random set of pairs of benchmarks were chosen from the NAS parallel benchmark suite.

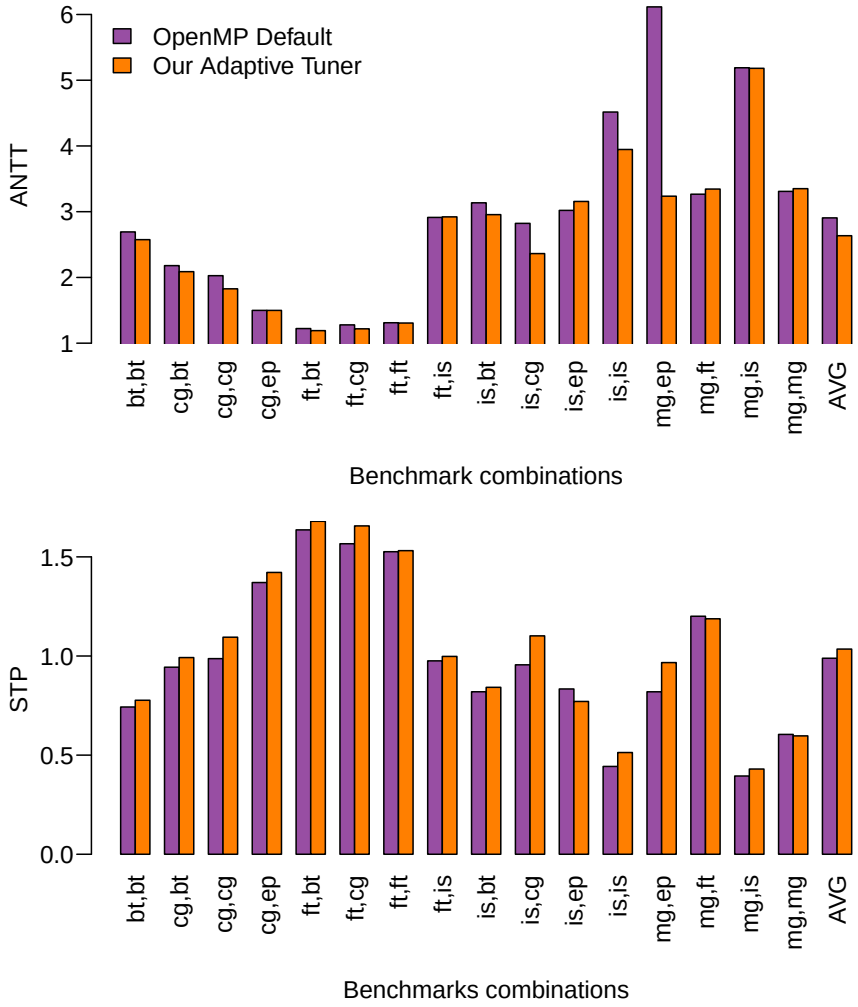

Figure 18: Plot showing the ANTT and STP achieved by ThreadTuner-OMP, the scalability-based thread count tuner compared to the libgomp implementation of OpenMP. For ANTT, lower is better. For STP, higher is better. Each benchmark combination is run multiple times, until the coefficient of variation of the execution times drops below 0.01.

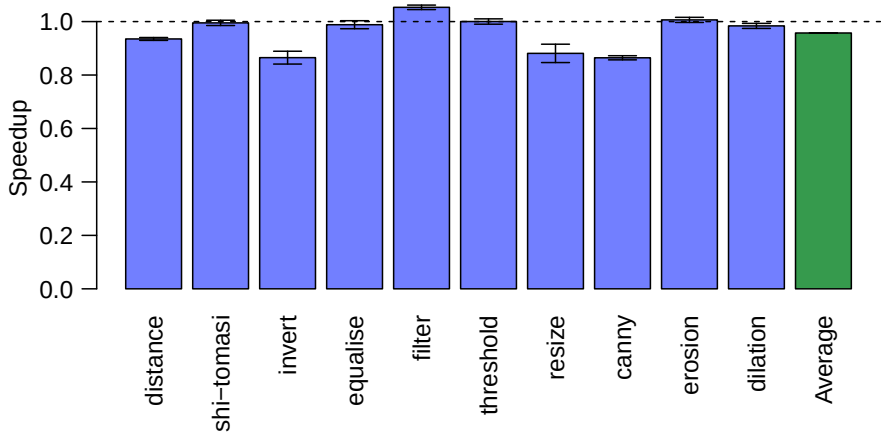

Figure 19: Overhead of the runtime monitor for a range of image processing benchmarks. Reconfiguration messages were sent to the programs at 100ms intervals. The benchmarks are each run repeatedly until the coefficient of variation in the results drops below 0.01. Error bars show 99% confidence intervals for the mean.

These programs were then run concurrently using both the default OpenMP scheduler and our adaptive scheduler. The execution time of each program was measured and used to compute the ANTT of the system, therefore lower is better.

These results demonstrate that, on average, our approach achieves better performance than the OpenMP default. In one case our approach did as well as, but not better than, the default OpenMP scheduler. This is likely due to the slight overhead incurred by the inter-process communication performed by our scheduler.

4.7.3 *Runtime Monitor Overheads*

This section presents an empirical evaluation of the overheads introduced by communication between programs and the runtime monitor. Repeated reconfigure messages are sent to the programs with a fixed resource allocation. We measure the total execution time of a range of image processing benchmarks, with and without the runtime monitor code enabled. Each of these programs is run in isolation.

Figure 19 shows the slowdown incurred by the runtime monitor for each of the image processing benchmarks. The baseline is our multithreaded C++ implementation without the runtime monitor component. The runtime monitor was configured to send a reconfigure message to each program continuously, at 100ms intervals. The average slowdown was 0.95x. This means that we need a performance improvement of just 5% in order to overcome the overhead incurred by the extra communicaton.

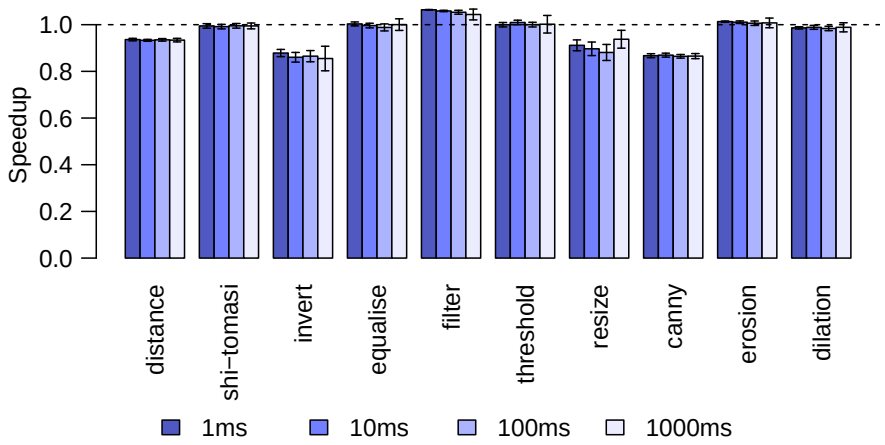

Figure 20: Overhead of the runtime monitor for a range of image processing benchmarks, for a variety of reconfiguration message intervals. The benchmarks are each run repeatedly until the coefficient of variation in the results drops below 0.01. Error bars show 99% confidence intervals for the mean.

The results also show that the slowdown varies across programs. For 5 of the 10 benchmarks performance was not affected. 4 of the program experience about 10% degredation in performance. Interestingly, the filter benchmark exhibits a *speedup* of 8%. We are investigating what is causing this counter-intuitive result.

These experiments were run on a 32-core shared memory machine.

4.7.4 *Sensitivity Analysis*

We ran a further experiment to determine how performance is affected by the frequency of reconfigurations. Figure 20 shows the slowdown for each benchmark using 4 intervals for the reconfigure messages, ranging from 1ms to 1000ms. The execution time of each benchmark is approximately 0.5 seconds, so using 1000ms as the interval means that no reconfigure messages are sent to the program.

The results show that the overhead is unaffected by the frequency of the reconfiguration messages. There is a fixed overhead to be paid for use of the runtime monitor, but the additional overhead introduced by the increased number of reconfigurations is negligible. This shows that the overhead incurred by our implementation scales well to a high-frequency of reconfigurations. Such a scenario would present itself when the workload of the system changes rapidly.

4.8 SUMMARY

In this chapter, the need for scalability-based tuning of thread counts was motivated. A tuning technique for dynamically choosing the thread counts for skeleton parallel programs was developed, based on scalability information collected a priori for each program. ThreadTuner-OMP achieves an ANTT of 2.63 and an STP of 1.03 which, compared to the libgomp OpenMP implementation, is an 11% improvement in ANTT and a marginal improvement in STP.

The next chapter explores the second of the three cooperative auto-tuning challenges addressed in this thesis: choosing where to schedule threads for parallel programs running on a multi-socket shared-memory system.

5

COOPERATIVELY TUNING THREAD PLACEMENT

The previous chapter explored how to cooperatively tune the number of threads used by programs, in order to improve overall system performance. However, it assumed that the system consists of a single socket. This chapter addresses the second challenge of cooperatively tuning: where should each programs thread be executed?

Different program-to-core mappings for a dual-socket machine are explored, and their performance impact measured. From this data we devise a spatial-scheduling heuristic, named LIRA, for selecting which programs threads should run be scheduled to run on the same socket. From this heuristic, two flavours of scheduler are implemented: (*i*) LIRA-Static collects performance data in an offline profiling step to decide the schedule when a program starts its execution, and (*ii*) LIRA-Adaptive which operates dynamically at runtime, using online hardware performance counters data to adapt the schedule during program execution.

Section 5.2 motivates the need for this work by showing the performance impact of different program-to-socket mappings on the ANTT and STP of the system. These performance metrics are described in detail in Section 2.4.1. Section 5.3 explores the performance degradation caused by ignoring the presence of multiple sockets, and devises the LIRA heuristic that is used predict the best program-to-socket schedule. Section 5.4 describes the design and implementation of the static and dynamic schedulers, LIRA-Static and LIRA-Adaptive. Section 5.5 evaluates the performance gains provided by these schedulers compared to two competing approaches: Callisto (Harris et al., 2014) and the libgomp OpenMP runtime systems. We demonstrate that LIRA-Adaptive is better than the offline static approach LIRA-Static that uses the same heuristic. We demonstrate improvement in both STP and ANTT. Finally, Section 5.6 summarises the chapter, including a discussion of the limitations of this approach.

5.1 INTRODUCTION

The previous chapter explored how to cooperatively tune the number of threads used by programs. Based on the scalability of each program, a number of threads was assigned to each program running on the system. However, this makes the assumption that every core is equal, and the spatial allocation of threads to cores was simple: pin each thread to its own core. This chapter addresses the challenge of where to schedule the program threads that are executing on the system.

In multi-socket machines it is not the case that each core is equivalent. Firstly, communication between cores on different sockets is more expensive than between cores in the same socket, as it must cross the inter-socket interconnect. Secondly, each socket has a dedicated memory controller and main memory. Therefore sharing memory between cores on different sockets incurs additional memory traffic to copy the data between sockets. The presence of multiple memory controllers also provides more bandwidth for memory operations, if used correctly. By balancing the load on the memory controllers by scheduling the programs to different sockets, the system memory controller resources can be fully utilised.

In this chapter, we explore running a dynamically changing mix of parallel programs on a multi-socket shared-memory machine. We explore how to make online decisions about which of these programs' threads should be co-located on the same single socket.

An online adaptive scheduler, named LIRA-ADAPTIVE is developed, that selects which sets of programs should share cores on the same socket, and is able to respond to phase changes within a program's execution.

We build on Callisto (Harris et al., 2014), a user-mode framework for prototyping schedulers and exploring the interaction between the system-wide scheduler and the runtime systems in individual programs. Callisto reduces scheduler-related interference between sets of programs running together, but ignores memory system interference present in multi-socket NUMA machines. This work is discussed in more detail in Section 3.2.1 and Section 5.4.1.

We evaluate LIRA-ADAPTIVE by comparing it with: (*i*) BEST-STATIC which selects the best program-to-socket mapping for a given workload by exhaustively trying every combination a priori, (*ii*) LIRA-STATIC which selects the program-to-socket mapping using the same heuristic as LIRA-ADAPTIVE but based on per-program solo-run profiling, (*iii*) Callisto and (*iv*) libgomp from GCC with scheduling performed by

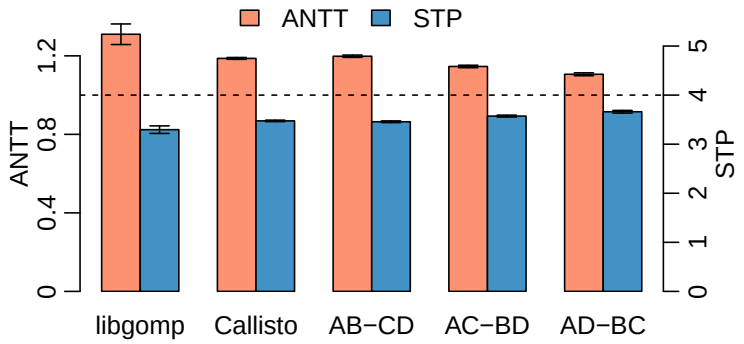

Figure 21: The Average Normalized Turnaround Time (ANTT) and System Throughput (STP) achieved by: (*i*) libgomp from GCC 4.8.0, (*ii*) Callisto and (*iii*) Callisto with the three possible static program-to-socket mappings. The programs are run concurrently on a dual-socket machine, with at least 9 repeats. For ANTT, lower is better, and for STP, higher is better. Error bars show 95% confidence intervals for the mean.

the default Linux OS scheduler. LIRA-STATIC avoids the cost of collecting profiling information during execution, but also prevents adaptation to phase behavior.

5.2 MOTIVATION

Figure 21 shows the performance of running four programs concurrently on a dual-socket machine with 8 cores per socket, using different runtime systems and program-to-socket mappings. The experiment uses AMMP (A) and SWIM (B) from the SPEC OMP 2012 benchmark suite, and PAGERANK (C) and TRICOUNT (D) from the Green-Marl domain-specific graph analytics project (Hong et al., 2012). The programs all run concurrently. Each is run for at least 9 repeats, and possibly more to ensure that all 4 programs are running throughout the experiment. We measure two system-wide performance metrics: ANTT and STP. These metrics are defined and discussed in more detail in Section 2.4.1.

Figure 21 compares five scheduling variants for this workload:

LIBGOMP The OpenMP implementation of libgomp from GCC 4.8.0. Each program is run using a separate instance of the libgomp library. Each is configured to use passive synchronization and 16 OpenMP threads each. Therefore, each program has sufficiently many threads to make use of all cores on the system. Scheduling of these threads is performed by the default Linux 2.6.32 scheduler.

CALLISTO This variant uses Callisto (Harris et al., 2014). Each program creates 16 OpenMP threads, and Callisto multiplexes these over the 4 cores allocated to each program. The sets of 4 cores will generally be in the same socket, but there is no control over specifically which program gets which set of cores (this may vary over time as programs start and complete). If one program cannot use all of its allocated cores, then Callisto makes these available to other programs; this provides AMMP and SWIM with additional cores when PAGERANK and TRICOUNT are loading their input graphs.

AB-CD
AC-BD
AD-BC These three configurations use a modified version of Callisto that fixes each program's threads to a specific quarter of the machine. For programs A–D, the notation indicates which pairs of programs are placed on the same socket. For instance AD-BC indicates that A and D are together, and that B and C are together. Accounting for symmetry, there are three alternative choices of allocating programs to sockets.

The results show that the choice of program to socket mapping has an impact on both the ANTT and STP of the system. Callisto achieves better ANTT than libgomp OpenMP, which is expected given that Callisto's aim is to reduce interference between programs. This interference can be reduced further by partitioning programs within separate sockets, as shown by the improved ANTT and STP achieved by each of the configurations AB-CD, AC-BD and AD-BC. Moreover, the choice of pairings of programs has a significant effect on performance. The ANTT and STP varies amongst the three configurations, with AD-BC achieving better ANTT and STP compared to the other two. LIRA-ADAPTIVE attempts to identify these best pairings at runtime.

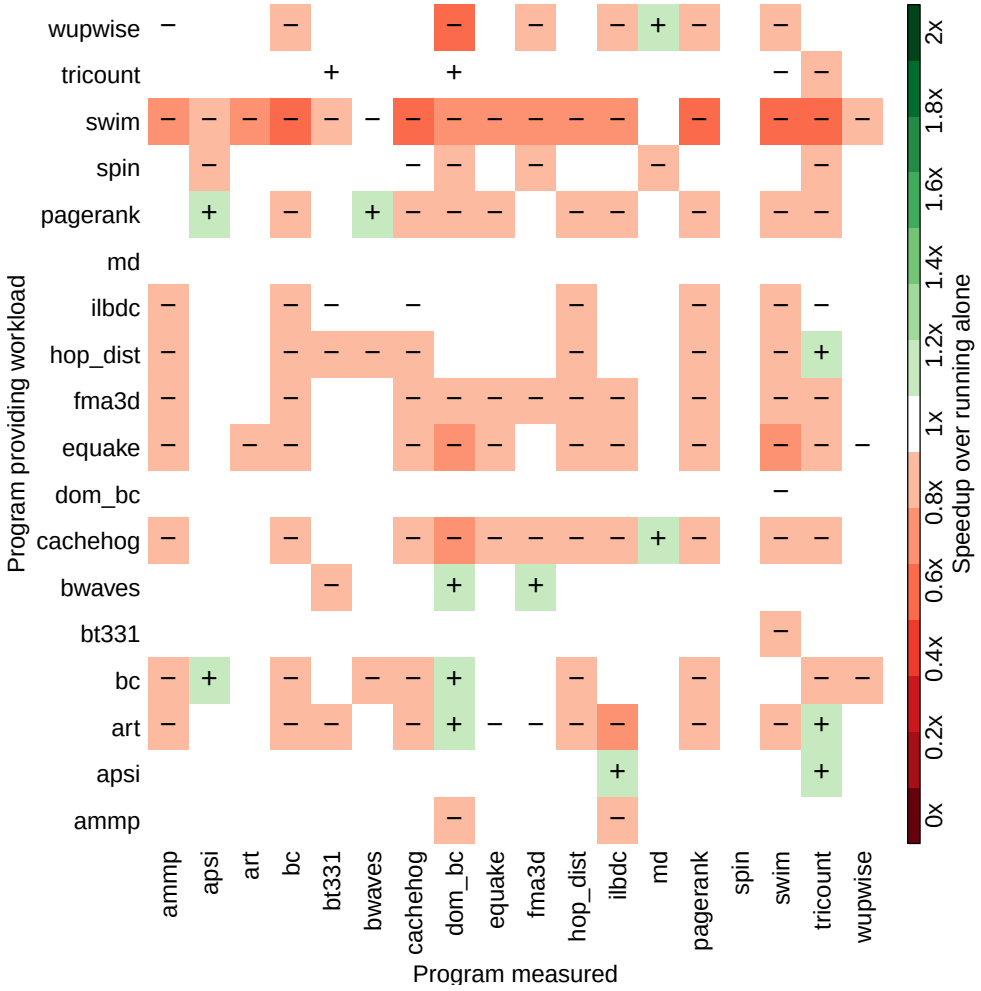

Figure 22: Pair-wise speedup of programs, comparing sharing a socket to using separate sockets. Boxes annotated with a – indicate cases where performance decreased, and + where performance increased. Regions marked with a – show where system performance can be improved.

5.3 SOCKET SCHEDULING HEURISTIC

In this section we describe how LIRA characterises programs at runtime, to identify a schedule that is likely to perform well. Section 5.3.1 explores the performance degradation that occurs when running pairs of programs on a multi-socket machine, and Section 5.3.2 describes our heuristic technique for predicting program pairings that minimize this degradation.

5.3.1 *Pairwise Performance Degradation*

Figure 22 shows a comparison between running pairs of programs on the same socket to running them on distinct sockets. We use a 16 core dual-socket machine for this experiment. The programs used are detailed in Section 5.5.1. Each program is configured to run 4 threads, pinned to either 4 distinct cores on different sockets (A_-B_ using our previous notation), or 4 distinct cores on the same socket (AB-__). This setup ensures that each program is given the same amount of computational resource – 4 threads pinned to 4 distinct physical cores – and that each thread has exclusive use of the core to which it is pinned. Therefore any change in performance is due to thread placement.

These results show that there is a significant performance penalty associated with sharing the a socket with another program. For about half of the program combinations there is an increase in execution time of 20%, and a maximum increase of 50%. For the other half of the programs there is minimal impact on sharing sockets. In some rare cases, there is actually an increase in program performance. In two cases, there is a 1.4x increase in performance. See Section 5.5.2 for further discussion of this.

These results suggest that a smarter scheduling approach could avoid program slowdown by carefully choosing which programs should share the same socket.

It is also interesting to note that there are clear rows of red for some applications and columns of white for others. For example, MD is amenable to sharing the system as it does not experience or cause slow-down when paired with any programs. In contrast, SWIM adversely affects the performance of most programs when they run alongside it, except for MD. Therefore MD and SWIM are good candidates to co-locate on the same socket, to improve overall system performance.

5.3.2 *LIRA: Heuristic for Socket Scheduling*

Modern CPU architectures provide many hardware performance counters, in the form of a set of dedicated hardware registers that are incremented by the control logic of the CPU itself. These can be used to record events on a per-thread basis, with low overhead or impact on the behavior of the program. They include events such as the number of cache misses at different levels of the hierarchy and the number of completed instructions. The specific events that are available is dependent on the underlying hardware. We use the Performance API library (Mucci et al., 1999) to set

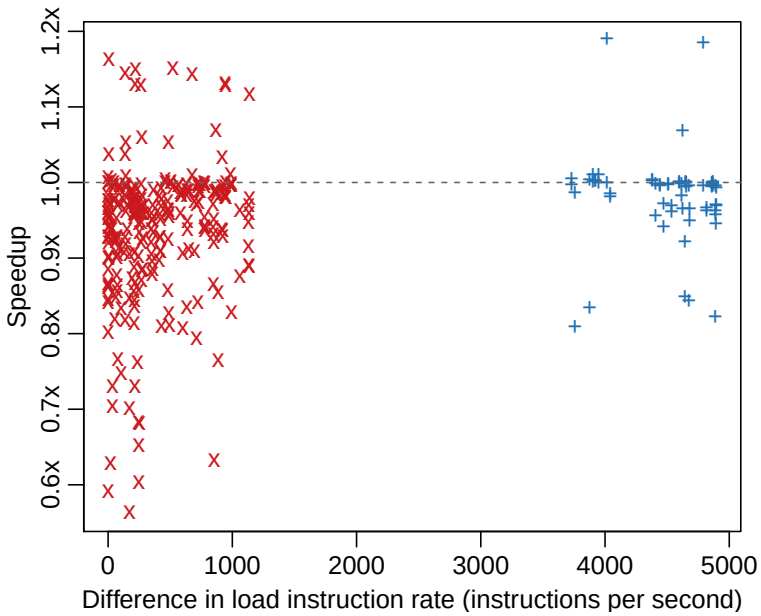

Figure 23: Average speedup of pairs of programs compared to execution in isolation, plotted against the absolute difference in the load instruction execution rate. There are two distinct clusters in the data, shown by the red x and blue + markers. The dashed line shows a speedup of 1×.

up and measure hardware counters on our experimental platform. These counters provide some measure of the behavior of programs, that we can use as program features to build our predictive model which is the basis of our scheduling heuristic.

Figure 23 shows the speedup of running pairs of programs concurrently on the same system over running them in isolation, against the absolute difference in the rate at which the programs execute load instructions. This rate is measured separately for each of the program's threads then averaged using the arithmetic mean. For these measurements, we use the same 16 core dual-socket machine as before. These results show that the average slowdown is greatly reduced when the difference in load instruction rate is maximized. Program pairs with a large difference in load instruction rate (the cluster to the right of the plot) have a geometric mean speedup of 98% and a maximum slowdown of 19%. In contrast, program pairs with a small difference in load instruction rate (the cluster to the left of the plot) have smaller geometric mean speedup of 93% and a much higher maximum slowdown of 44%. This shows that pairs of programs with different load instruction rate are more likely to achieve good performance. The intuition here is that, by pairing programs in this way, pressure on the memory system is reduced. We use this as our heuristic for predicting which pairs of programs will cooperate more effectively when run on the same socket.

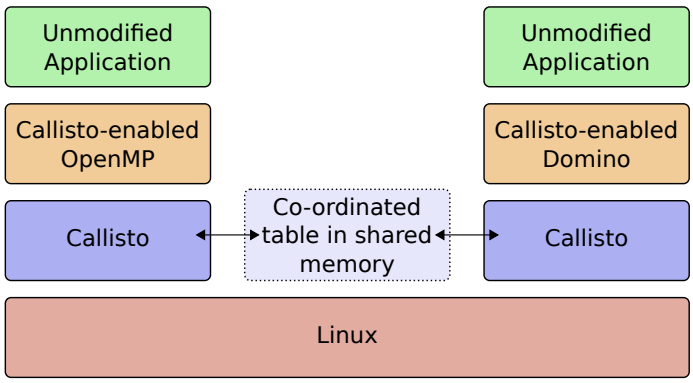

Figure 24: Structure of a system using Callisto. Reproduced from Harris et al. (2014)

THE LIRA HEURISTIC Given a set of programs to run on the system, each with a given load instruction rate, we choose a mapping from programs to sockets such that the absolute difference in the instruction rate of the programs on each socket is maximized. Consider a dual socket machine with four programs running, programs A and B are scheduled to the first socket, and programs B and C to the second socket. The programs have load instruction rates R_A, R_B, R_C and R_D. The chosen schedule is the one that maximises the following expression:

$$\text{abs}(R_A - R_B) + \text{abs}(R_C - R_D) \tag{11}$$

5.4 SPATIAL SCHEDULING FOR SOCKETS

In this section we introduce LIRA-ADAPTIVE, an online adaptive scheduler built on top of Callisto Harris et al. (2014). Section 5.4.1 explains how Callisto's spatial thread scheduling works. Section 5.4.2 discusses Callisto's behavior in a multi-socket environment. We then describe two variants of our multi-socket-aware scheduler: Section 5.4.3 describes LIRA-STATIC which uses profile data to perform static scheduling, and Section 5.4.4 describes LIRA-ADAPTIVE which performs online adaptive scheduling. These schedulers build on Callisto to improve thread to core scheduling in a multi-socket system, using the LIRA heuristic described in Section 5.3.2.

5.4.1 *Callisto's Thread Scheduler*

The Callisto runtime system uses *dynamic spatial scheduling* to allocate threads to physical cores. The structure of the Callisto runtime system is shown in Figure 24.

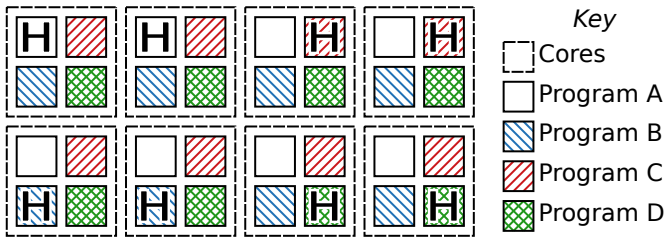

Figure 25: Example of Callisto's spatial scheduling. High priority threads are marked with an **H**. Each program is given an equal number of high priority threads.

Each program that runs on the system spawns multiple *worker* threads and pins each thread to each physical core. Of the threads pinned to each core, one is designated the *high priority* thread, and the remainder as *low priority* threads. An example of this is shown in Figure 25. Callisto ensures that each program has an equal share of high priority threads, and that the main thread for each program is given high priority. The aim of this is to ensure that the main thread can always run, as it often acts as a producer of parallel tasks, and so its performance is critical to the performance of the program as a whole. This also provides a fair distribution of resources across all running programs.

Callisto's aim is to run high priority threads most of the time. This means that the high priority threads experience low interference from other threads running on the system. For example, they can make full use of core-local caches, without the threat of other programs evicting cache lines that would lead to performance degradation. This setup also reduces the number and frequency of context switches, reducing the overhead they incur.

In order to maintain good utilization of resources, a low priority thread is allowed to run when the high priority thread is not runnable, for example when the high priority thread blocks for disk accesses or synchronization. Due to the bursty nature of many parallel workloads, which is also true of many of the benchmarks used in our evaluation, this is essential to make good use of the available hardware resources. Callisto limits the frequency with which context switching to low priority threads can occur using a configurable hysteresis threshold, typically around 10ms. If a high priority thread blocks for longer than a fixed number of processor cycles, it is stopped and a low priority thread allowed to run. The high priority thread is only allowed to run again after it has been runnable for sufficiently many processor cycles.

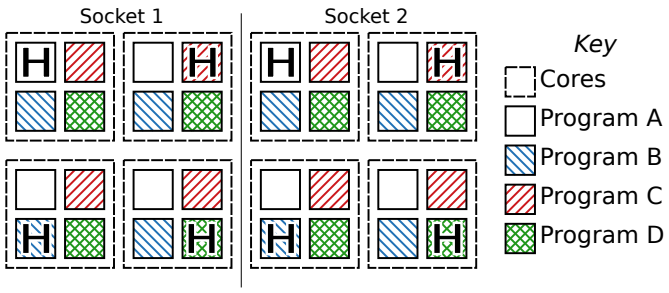

Figure 26: Example of a pathologically bad schedule produced by Callisto's spatial scheduling. High priority threads are marked with an **H**. Each program is given an equal number of high priority threads, however they are placed on separate sockets. This will cause maximal inter-socket communication overheads.

5.4.2 *Multi-Socket Scheduling*

Callisto's thread scheduler treats the system as a homogeneous array of cores. It arbitrarily assigns programs to cores, and allows a program to have threads running on different sockets. This means it does not necessarily allocate programs to sockets in a manner that reduces interference. Callisto's spatial scheduler can lead to the situation where a low priority thread is run on a different socket from the high priority threads. This is likely to incur additional inter-socket communication as data is copied to the caches on the other socket. Synchronization may also have to be performed across the socket boundary in this case, which may cause the high priority threads to block whilst waiting for the low priority thread to complete. For example, Callisto could produce the pathologically bad schedule shown in Figure 26. Each thread needs to communicate across the socket boundaries, which introduces communication overheads that would be unnecessary if the threads were scheduled to the same sockets.

Our scheduler uses the LIRA heuristic to extend Callisto by considering the fact that the cores exist in separate sockets. Our approach aims to automatically allocate programs to sockets such that interference and contention for resources is reduced. When there are more programs than sockets, we also keep all of the threads for each program on the same socket, to avoid the situation where a low priority thread is run on a separate socket. However, within each socket, we use Callisto to schedule the threads. This improves utilization within each socket, by allowing low priority threads to run if the high priority threads block.

We develop two scheduling techniques: LIRA-Static (Section 5.4.3) uses profile data collected a priori to decide which programs to allocate to which sockets, and

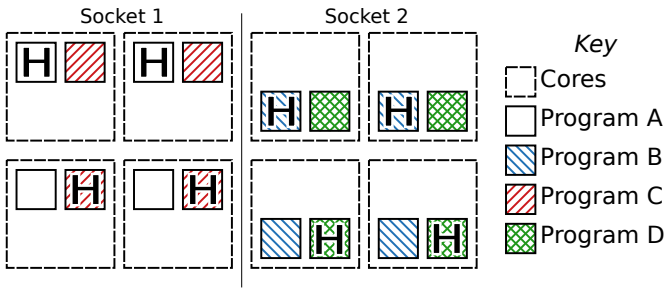

Figure 27: An example of a schedule produced by LIRA-STATIC, our profile-driven scheduler. Program pairings A,B and C,D are determined to be optimal using the scheduling heuristic in Section 5.3.2.

LIRA-ADAPTIVE (Section 5.4.4) observes the programs at runtime to adaptively allocate programs to sockets.

Our profile-based and online adaptive approaches rely on being able to anticipate when running programs on the same socket will lead to bad performance, compared to running them on separate sockets, and avoid these cases. In order to do this, we devise a model that maps the properties of the pairs of programs to a performance estimate. We use hardware performance counters to provide these properties.

5.4.3 *LIRA-Static: Profile-Driven Scheduler*

Our profile-driven scheduler uses information about program behaviour collected *a priori* to schedule programs to sockets.

To run a program on the system, it must first be profiled. The application programmer provides a sample input and the program binary to the system. The program is then run exclusively on a single socket of the machine, and hardware performance counters are used to measure its behaviour. The values of these are converted to rates (normalized by the total execution time of the program) and stored in a database for use in scheduling decisions.

When a program is run on the system, the scheduler examines the database for the behaviour data for every program that is running. This data is used to predict the best allocation of *programs* to sockets. We then use the original Callisto strategy to schedule each program's threads within each socket. This means that, within each socket, each program spawns and pins one thread to every core, and each program has an equal share of high priority threads. This prediction is made using the LIRA heuristic described in Section 5.3.2. Figure 27 shows an example of the spatial scheduling performed.

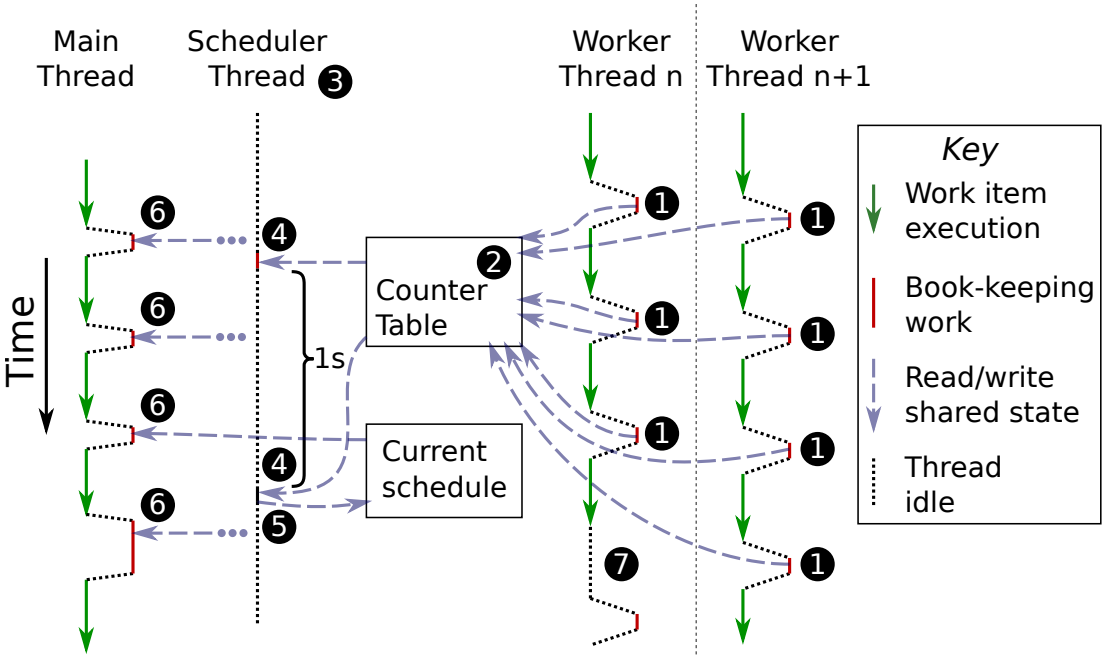

Figure 28: Operation of LIRA-Adaptive, the online adaptive scheduler.

This scheduling decision is only made when a program is invoked. The schedule does not adapt during program execution, or program phases. In our experiments we focused solely on the case where sets of programs are run simultaneously.

This static profile-driven approach requires a potentially expensive training phase, however it incurs no runtime overhead. This approach is used as an additional baseline to compare against our more sophisticated online adaptive scheduler, described in the following section.

5.4.4 *LIRA-Adaptive: Online Scheduler*

Our online adaptive scheduler uses the same LIRA heuristic as the profile driven scheduler, described in Section 5.4.3. This heuristic is used to choose the best program to socket allocation during program execution. This removes the need for a separate profiling step, makes the approach input agnostic, and allows the schedule to adapt to changes in program behavior during program execution.

The online adaptive scheduler consists of the following two components. Figure 28 shows a timing diagram of the operations performed by these components.

PERFORMANCE MONITORING Each program thread periodically measures its hardware performance counters (shown by ❶ in Figure 28), and updates a process-

shared table with this information (❷). The time interval between updates to this table is configurable. A sensitivity study of this parameter is presented in Section 5.5.4.

Each thread measures the number of executed instructions and executed load instructions since the last update. These values are used to compute the rate at which load instructions are executed since the previous update, for the program as a whole by averaging across all threads. Each thread also measures the number of CPU cycles spent in a runnable state since the last update. A thread is in the runnable state if it is a high priority thread and is not blocked for I/O or synchronization. This is used to determine whether a thread is idle as described in Section 5.4.4.1.

The process-shared table stores these hardware performance counter measurements for each thread running in each process on the system. The cache miss rates and number of cycles are smoothed using an exponential moving average. This smoothing avoids short lived changes from affecting the scheduling decision, which would incur large overheads due to frequently moving threads to different cores.

SCHEDULING Each program spawns an additional thread to perform scheduling decisions (❸). These threads are pinned to the same cores as each program's main thread. These scheduler threads periodically check the information stored in the shared-process table (containing performance information collected by the performance monitor) to decide if the thread schedule should change (❹). The time interval between these updates is configurable. A sensitivity study for this parameter is presented in Section 5.5.4.

The scheduler computes the arithmetic mean of the cache miss rates for each thread in each program. It then uses the LIRA heuristic (described in Section 5.3.2) to assign a numeric score to every possible placement of programs on sockets. The schedule with the highest score is then chosen as the new schedule.

This decision is written to a process-shared piece of memory (❺), so that the main threads in other programs can detect the change and apply the new schedule (❻). Work is allocated to threads based on the schedule, and threads that are not allocated work simply remain idle ❼.

5.4.4.1 *Dealing with Bursty CPU Load*

Programs often involve an input or output phase which requires significant amounts of I/O to disk. For example, the graph analytics benchmarks load a large graph from disk before performing computation over it. During these phases, CPU usage is min-

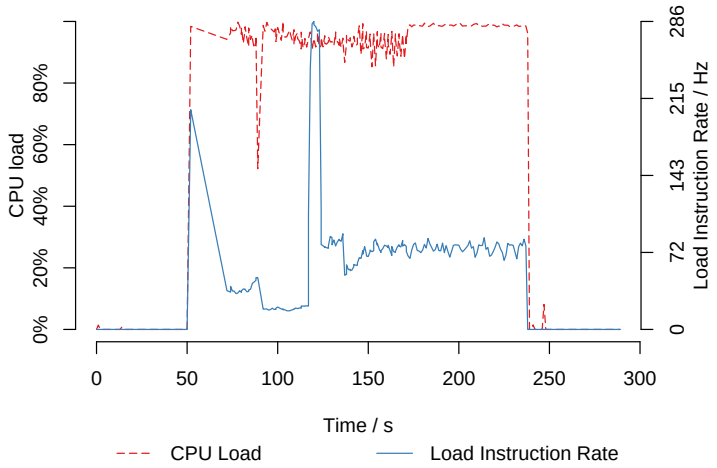

Figure 29: Trace showing the CPU load (red, dashed line) and load instruction rate (blue, solid line) over time for the PAGERANK program. This demonstrates the initial loading and completion phases with minimal CPU load. Phase changes in load instruction rate during high CPU load periods can also be seen.

imal – the worker threads are essentially idle. This makes the load instruction rate heuristic meaningless. The threads do not run very often therefore there no meaningful load instruction rate to measure. Figure 29 shows an example of this behaviour for the PAGERANK program.

LIRA-ADAPTIVE handles this by first classifying the programs running on the system as idle or active. This is done by measuring the number of cycles that each high priority threads spends in a runnable state (i.e. when not blocking for I/O or synchronization). This is converted to a percentage CPU load and averaged across all program threads. If this average CPU load is below 50% the program is considered idle. Note that the choice of this threshold is not important as the common case is that the program's average CPU load is either near 100% or near 0%.

The schedule is determined based on the following cases:

NO IDLE PROGRAMS The load instruction rate LIRA heuristic is used to schedule programs to sockets. This is the common case, when none of the programs are performing large amounts of I/O to disk.

AT LEAST 1 IDLE PROGRAM The load instruction rate of the idle program is assigned a value of 0. This will result in the program with the highest load instruction rate being scheduled to the same socket as the idle program. Also, the idle programs are not moved to a different socket.

MORE IDLE PROGRAMS THAN SOCKETS The schedule is left unchanged.

5.5 EVALUATION

In this section we analyse the performance of our profile-driven and online adaptive schedulers compared to the libgomp OpenMP implementation and Callisto. Section 5.5.1 details the setup for our experiments, Section 5.5.2 compares our scheduling techniques against OpenMP and Callisto, and Section 5.5.3 compares against an optimal static policy.

5.5.1 *Experimental Setup*

We use two performance metrics to compare and contrast the scheduling approaches – ANTT and STP. These metrics are described in detail in Section 2.4.

For our experiments we use a dual-socket machine, with a pair of Xeon E5-2660 processors clocked at 2.20GHz. Each processor has 8 physical cores with 2 hardware threads per core. We disable hyperthreading to focus on the effects of multiple sockets, and will investigate the effect of hyperthreading in future work. Each socket has 128GB of main memory, for a total of 256GB, and runs Linux 2.6.32. We use GCC 4.8.0 to compile the benchmark programs.

We use 18 benchmark programs taken from four different sources. Firstly, we use the 11 benchmarks from SPEC OMP 2001 and 2012 that are supported by Callisto. These are the benchmarks that do not use manual locking via calls to `omp_set_lock` and `omp_unset_lock`. These calls are used by (*i*) nested parallel sections, (*ii*) the ordered directive and (*iii*) explicit tasks.

We also include an implementation of the betweenness-centrality graph algorithm (Brandes, 2001) written using CDDP, a constrained data-driven parallelism programming model (Harris et al., 2013). Four graph analytics programs are also included. They are written in the domain specific Green-Marl language (Hong et al., 2012), which is compiled to OpenMP using the Green-Marl compiler.

Finally, we include a pair of micro-benchmarks: `spin` and `cachehog`. `spin` simply executes CPU-bound computation, with a very small working set, so as to put minimal stress on the memory system. `cachehog` executes memory-bound computation, with the aim of maximising the number of misses in the last-level cache. The intention of this is to provide a benchmark that makes the heaviest possible utilisation of the memory system. Running these synthetic workloads alongside other programs helps us understand the reasons for the behavior that we see.

Table 1: The 32 random program combinations used to evaluate the system

#	Programs			
1	ammp	bwaves	dom_bc	fma3d
2	ammp	ilbdc	md	tricount
3	ammp	pagerank	swim	tricount
4	ammp	bwaves	dom_bc	ilbdc
5	ammp	fma3d	spin	tricount
6	ammp	dom_bc	equake	pagerank
7	ammp	apsi	hop_dist	swim
8	apsi	art	equake	md
9	apsi	cachehog	hop_dist	md
10	apsi	cachehog	dom_bc	wupwise
11	apsi	art	bwaves	cachehog
12	apsi	cachehog	ilbdc	wupwise
13	apsi	art	bwaves	hop_dist
14	apsi	cachehog	spin	wupwise
15	apsi	bwaves	dom_bc	ilbdc
16	apsi	bc	ilbdc	pagerank
17	art	cachehog	ilbdc	wupwise
18	art	dom_bc	hop_dist	spin
19	art	dom_bc	fma3d	tricount
20	art	bwaves	md	spin
21	art	dom_bc	hop_dist	wupwise
22	bc	cachehog	ilbdc	pagerank
23	bc	bwaves	hop_dist	wupwise
24	bc	bwaves	md	wupwise
25	bc	fma3d	md	pagerank
26	bt331	md	spin	swim
27	bwaves	spin	tricount	wupwise
28	bwaves	equake	hop_dist	wupwise
29	bwaves	cachehog	equake	fma3d
30	cachehog	dom_bc	equake	wupwise
31	fma3d	ilbdc	md	tricount
32	ilbdc	pagerank	spin	swim

We use input sizes that require approximately one minute of execution time when run alone on the machine. For the SPEC OMP 2006 benchmarks, we used the "large" inputs, and slightly reduced size inputs for the SPEC OMP 2012 benchmarks. For the graph analytics benchmarks we use a large Twitter graph with 42 million nodes and 1,500 million edges.

From this set of 18 benchmark programs, 32 random combinations of 4 programs are chosen. Table 1 shows the 32 chosen combinations, and the number used to refer to each in the graphs of results. We run each of the 32 sets of programs using each of the runtime environments: the libgomp OpenMP implementation from GCC, the Callisto runtime library, our static profile-driven scheduler (LIRA-Static) and our online adaptive scheduler (LIRA-Adaptive). We also measure the performance of every permutation of statically allocating programs to sockets, to provide best and worst case bounds for LIRA-Static (Worst-Static and Best-Static). To compute the ANTT and STP for each instance, we record the execution time for each program run in isolation on the machine. We therefore run each benchmark program in isolation, utilizing all 16 cores on the machine and the libgomp OpenMP implementation.

Note that using this experimental setup we are comparing our online adaptive scheduler, with its performance tracking instrumentation, against libgomp and Callisto without runtime instrumentation. Our results therefore include any runtime instrumentation overhead. These overheads are amortized by the increased performance and are quantitatively analysed in Section 5.5.4.

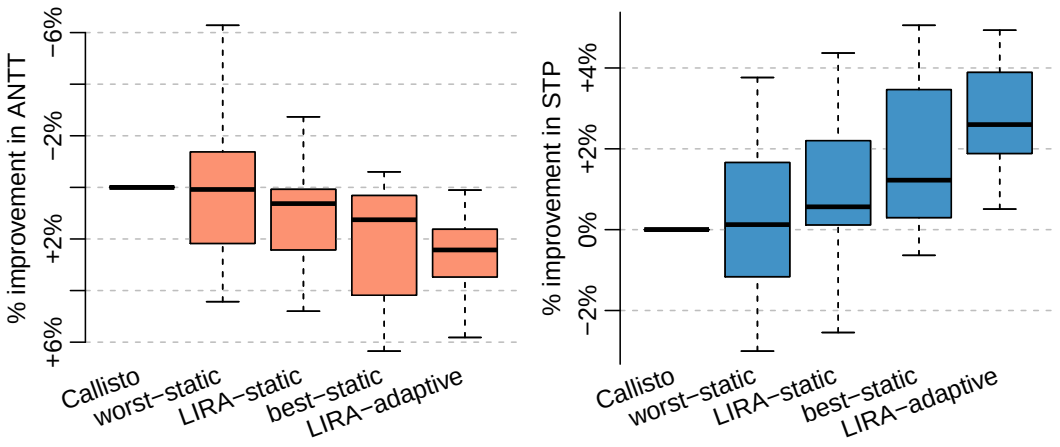

Figure 30: Box plots showing the percentage improvement in Average Normalized Turnaround Time (ANTT) (left) and System Throughput (STP) (right) achieved by each runtime system, across all 32 combinations of 4 concurrently running programs. For ANTT, lower is better, and for STP, higher is better. The thick horizontal lines show the median, the shaded boxes show the interquartile range, and the whiskers show the maximum and minimum values.

To quantify the error in our measurements, we run each program for at least 9 repeats. The running time of each benchmark program differs, therefore we repeatedly execute each of the four programs, until each of them have run for 9 repeats. This maintains the system workload for each program. We measure the time for the entire execution of each program, including loading data from disk.

5.5.2 *Comparison with libgomp and Callisto*

Figure 30 shows the percentage improvement in ANTT and STP achieved by each approach, compared to Callisto. This is computed as the percentage change in ANTT/STP for each program combination compared to the ANTT/STP achieved by Callisto.

Our results demonstrate that LIRA-Adaptive performs the best. In all cases it achieves performance at least as good as Callisto, and usually better. On average, LIRA-Static performs similarly to Callisto. This is likely due to both Callisto and the static approach choosing a single schedule for the entire program run. The schedule does not adapt to changes in program behavior during execution. This demonstrates that the adaptive approach used by LIRA-Adaptive is required. Moreover, any overhead incurred by LIRA-Adaptive's more complex implementation is amortized by the performance gains it provides.

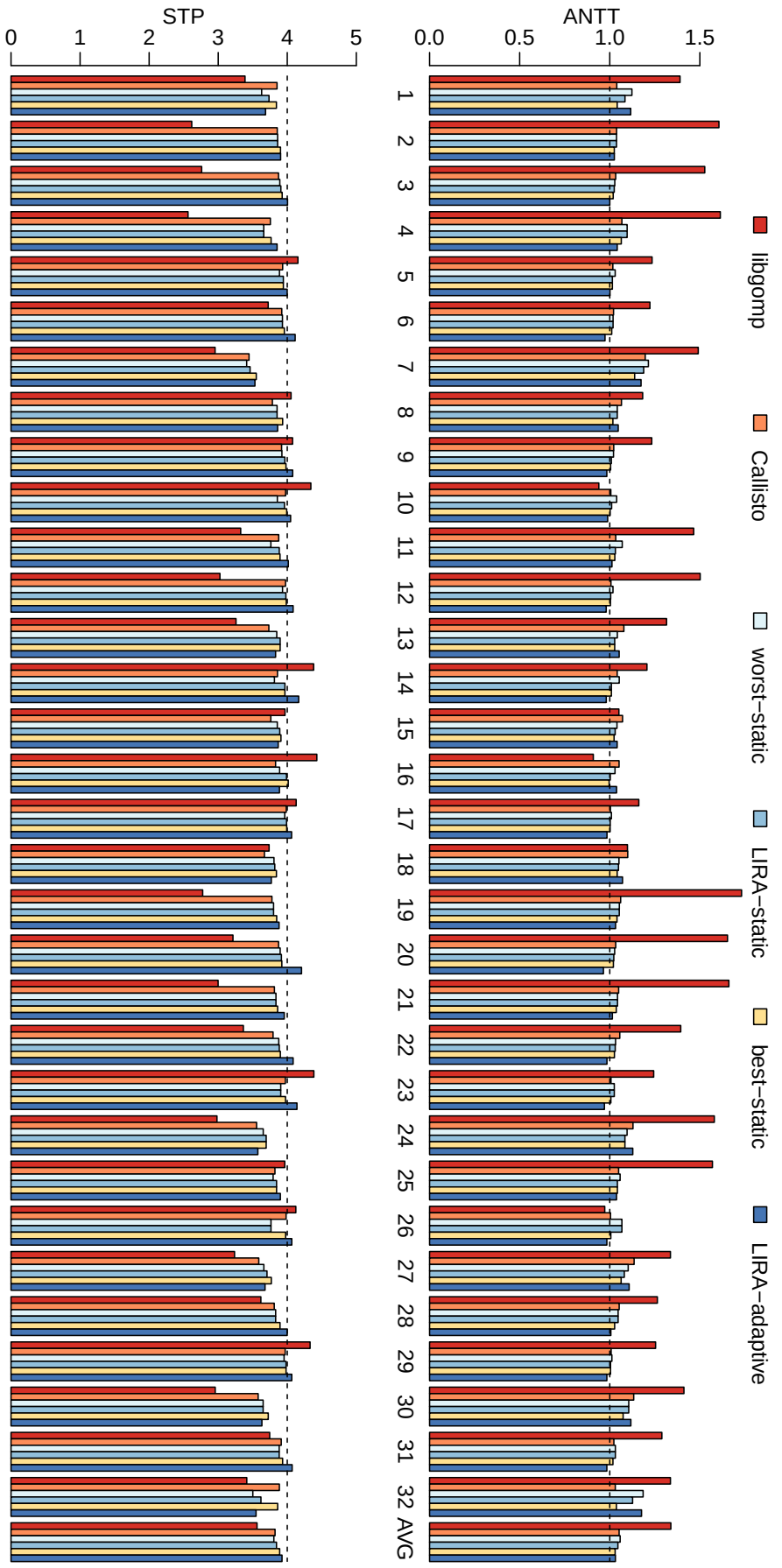

Figure 31: Comparison of the Average Normalized Turnaround Time (ANTT) and System Throughput (STP) achieved by different approaches, for 32 random combinations of 4 concurrently running programs. For ANTT, lower is better, and for STP, higher is better. AVG shows the arithmetic mean of the ANTT and STP for each approach.

Figure 31 shows the ANTT and STP for each program set achieved by both our static profile-driven scheduler (LIRA-STATIC) and online adaptive scheduler (LIRA-ADAPTIVE), compared to the libgomp OpenMP implementation, the Callisto runtime system, the best possible performance (BEST-STATIC), and the worst possible performance (WORST-STATIC), when choosing a schedule statically.

In most cases, libgomp performs the worst. This is not particularly surprising, as the benchmarks have likely been tuned under the assumption that the program will be using the machine exclusively. This demonstrates the need for socket aware scheduling given concurrently running programs.

In most cases, Callisto performs better than libgomp, achieving an ANTT close to 1 and an STP close to 4 in each case. LIRA-STATIC performs, on average, slightly better than Callisto. LIRA-ADAPTIVE performs similarly to the profile-driven approach for ANTT, improving performance in roughly half the cases, but degrading it in others. However, it significantly improves STP in many cases, achieving an STP greater than 4 in some cases. In some cases, LIRA-ADAPTIVE achieves an ANTT lower than 1. This shows that by pairing together programs that interact well performance can actually be improved compared to running in isolation. This is due to stalls in program execution being avoided by context switching to the other concurrently running program, but in such a way that the switch does not harm the performance of either program.

Three interesting cases are 10, 16 and 26. In these cases libgomp achieves the best ANTT. Callisto harms performance slightly, and LIRA-STATIC harms performance further. This is due to the increasing amount of runtime overhead introduced by each approach, and the fact that the programs in these cases do not interfere with one another (as shown in Figure 22). There is therefore no performance to be gained by carefully choosing the program-to-socket mapping, and therefore no performance gain that can be used to hide these overheads. In these three cases LIRA-ADAPTIVE also performs worse than libgomp, but at least as well as Callisto.

5.5.3 *Comparison with Optimal Static Policy*

Figure 31 also compares against two static oracles (BEST-STATIC and WORST-STATIC), which exhaustively try all static thread-to-socket allocations to find the best/worst choice. The results show that, in most cases, the profile-driven approach achieves the best performance it can (the static oracle provides an upper bound on the performance that a static scheme can achieve). This supports the hypothesis that the

LIRA heuristic is a good way of choosing non-interfering pairs. The results also show that in some cases there is large scope for improving performance using a dynamic approach. In some cases this comes at a cost in ANTT, but improves STP.

5.5.4 *Sensitivity Analysis*

Figure 32 shows the effects of varying the two implementation parameters present in LIRA-Adaptive: (*i*) the time delay between scheduler invocations which controls the frequency at which the system can adapt to changes in program behavior, and (*ii*) the time delay between samples of the hardware performance counter which controls the accuracy of the load instruction rate and CPU load information used to compute the LIRA heuristic.

This experiment was performed identically to previous experiments. We run 32 combinations of 4 concurrently running programs, on a dual-socket 16-core shared-memory machine.

TIME DELAY BETWEEN SCHEDULER INVOCATIONS For this parameter, we investigate time delays of 0.1 seconds (the value used by LIRA-Adaptive), 1 second and 5 seconds. We also compare against LIRA-Static, where the scheduler is invoked exactly once, at the start of the programs' execution, but with a priori knowledge of the hardware performance counters averaged over the program's entire execution. The time delay between samples of the hardware counters was set to 2×10^9 cycles for this experiment.

The top two plots in Figure 32 show the effect of this parameter on ANTT and STP. As the time delay is increased, performance decreases (shown by an increase in ANTT and a decrease in STP). The median ANTT and STP is similar for each plot, however the spread of ANTT and STP for the 32 program combinations increases. This increased spread of values shows that some programs performance is adversly affected by increasing this parameter value, whilst others maintain the same performance.

In the case of only running the scheduler at the start of the programs' execution, the performance across all benchmarks decreases significantly. This is shown by the increase in ANTT and decrease in STP. This demonstrates that best performance is achieved when the scheduler is invoked frequently, so that the runtime system can adapt the scheduler to the changing program behaviour.

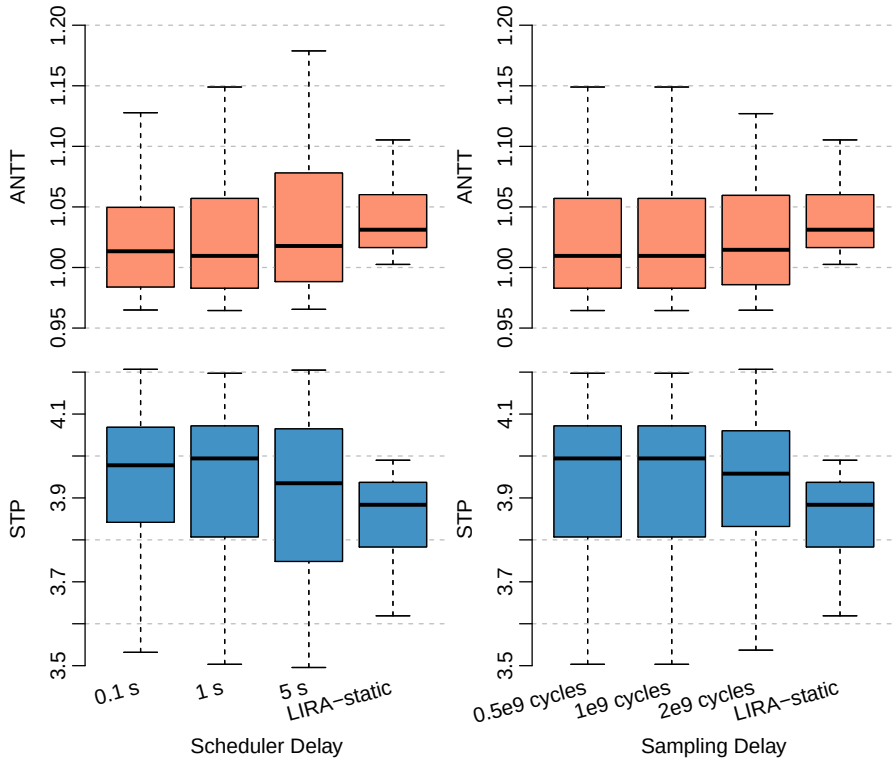

Figure 32: Box plots showing (*i*) the effect of varying the time delay between scheduler invocations on Average Normalized Turnaround Time (ANTT) (top left) and (*ii*) System Throughput (STP) (bottom left), and (*iii*) the effect of varying the time delay between samples of the hardware counters on ANTT (top right) and (*iv*) STP (bottom right). For ANTT, lower is better, and for STP, higher is better. The thick horizontal line shows the median, the box shows the interquartile range, and the whiskers show the maximum and minimum values. LIRA-ADAPTIVE uses a time delay of 0.1 seconds between scheduler invocations, and a time delay of $\mathbf{0.5} \times \mathbf{10^9}$ cycles between hardware counter samples.

TIME DELAY BETWEEN HARDWARE COUNTER SAMPLES For this parameter, we investigate time delays of 0.5×10^9 cycles (the value used by LIRA-ADAPTIVE), 1×10^9 cycles and 2×10^9 cycles. We also compare against LIRA-STATIC. The time delay between scheduler invocations is 5 seconds for this experiment.

The bottom two plots in Figure 32 show the effect of this parameter on ANTT and STP. Performance is similar for time delays of 0.5×10^9 and 1×10^9 cycles. Performance decreases for a time delay of 2×10^9 cycles, shown by an increase in the median ANTT, and decrease in the median STP. Performance is worst for LIRA-STATIC, where the scheduler is unable to respond to changes in program behaviour. This shows that frequent measurements of the hardware counters are required to provide the scheduler with accurate and timely information, so that it can adapt the schedule to changes in program behavior.

5.6 SUMMARY

In this chapter, the need for socket-aware scheduling was motivated. Both a profile-driven and online adaptive approaches: LIRA-STATIC and LIRA-ADAPTIVE. These schedulers map programs-to-sockets to reduce inter-socket interference and improve resource sharing in multi-program multi-socket systems. LIRA-ADAPTIVE does not require separate, offline workload characterization runs, and it accommodates a dynamically changing mix of applications, including those with phase changes.

LIRA-STATIC and LIRA-ADAPTIVE are evaluated using programs from SPEC OMP and two graph analytics projects. These two approaches are compared to the best possible performance obtained across all static mappings of 4 programs to 2 sockets, the libgomp OpenMP runtime that comes with GCC and Callisto, a state-of-the-art scheduler Harris et al. (2014). LIRA-STATIC improves system throughput by 10% compared to libgomp, and LIRA-ADAPTIVE improves system throughput by 13%. Compared to Callisto, LIRA-ADAPTIVE improves performance in 30 of the 32 combinations tested, with an improvement in system throughput of up to 7%, and 3% on average over 32 combinations.

6

COOPERATIVELY TUNING PARALLEL FRAMEWORK PARAMETERS

The previous chapter explored the challenge of how to cooperatively auto-tune the placement of program threads on sockets, in order to improve overall system performance. This chapter addresses the third and final cooperative auto-tuning challenge: how to cooperatively auto-tune the program implementations, by adjusting high-level parameters provided by the parallel framework.

In particular, this chapter presents work exploring the effect on overall system performance of the grain size parameter in Intel's Threading Building Blocks (Intel, 2012) when multiple parallel programs are running concurrently. Experiments are presented that demonstrate the need for cooperative tuning of this high-level implementation parameter. A tuning approach trained using an offline static tuner, similar in design to that presented in Chapter 4 is demonstrated to be ineffective in this scenario. The correlation between TBB performance counters and grain size is explored, and used to motivate a tuning approach which is evaluated using data collected from an exhaustive exploration of the optimisation space.

The rest of this chapter is structured as follows. Section 6.2 motivates the need for cooperative tuning of the grain size high-level implementation parameter present in Intel's TBB framework, when multiple programs are running on a shared system. Section 6.3 describes an approach trained using offline static information, and demonstrates that it is ineffective at tuning the grain size parameter in TBB. Section 6.4 explores the correlation between TBB performance counters and the best choice of grain size in order to improve system performance. Section 6.5 evaluates the performance improvement provided by using TBB performance counters to auto-tune the grain size of concurrently running programs. This approach is compared against the default TBB settings and the best possible performance found by an exhaustive search of the space of grain size parameter values. Section 6.6 summarises the work presented in this chapter, which is the final technical contribution of this thesis.

6.1 INTRODUCTION

In addition to the choice of how many threads to spawn and where to schedule them, addressed in the previous two chapters, we also require automatic tuning of implementation parameters exposed by parallel frameworks. These parameters directly affect the behaviour of parallel programs, including their resource usage characteristics and communication patterns. They therefore have a direct impact on the performance of parallel programs. Moreover, these parameters affect the performance of other programs running on the same system and need to be tuned in a cooperative manner.

In this chapter, we investigate parallel programs implemented using Intel's Threading Building Blocks framework (Intel, 2012). This library contains a *grain size* parameter which affects the granularity of the parallel programs running on the system. A smaller grain size causes a large number of small parallel tasks to be executed, whereas a larger grain size will result in a smaller number of large tasks. A small grain size therefore provides more parallel tasks and therefore larger scope for parallel execution, however it may lead to increased overheads such as synchronisation and inter-thread communication.

Programs implemented using TBB define the notion of task, and the grain size parameter controls how many of these tasks are coalesced before being executed by a worker thread. TBB programs are manually tuned to set this grain size parameter, or simply use the library default value of one. However, the optimal grain size is both dependent on the application and on the system workload, making the conventional methods of tuning the grain size sub-optimal. We therefore require a more intelligent approach for choosing values for the grain size parameter of each running program.

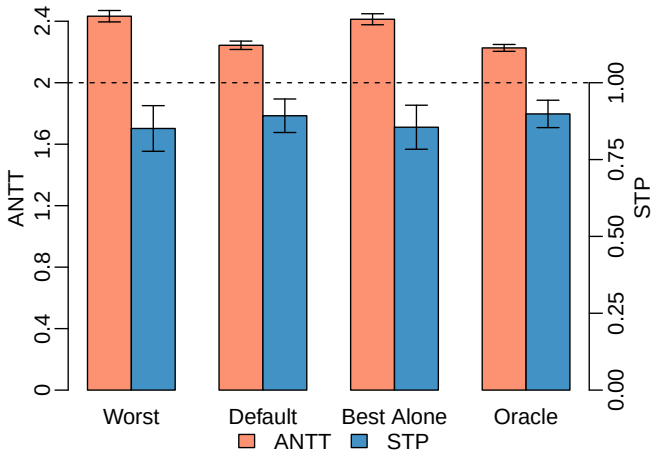

Figure 33: Performance achieved by each method of choosing the grain size parameter for the TACHYON and STREAMCLUSTER benchmarks implemented using Intel's TBB framework. System performance is reported using the ANTT and STP metrics. For ANTT, lower is better, and for STP, higher is better. The experiment was run on a 4-core shared memory multi-core machine. Each program is run multiple repeats until the coefficient of variation is below 0.01. Error bars show 99% confidence intervals for the mean.

6.2 MOTIVATION

Figure 33 shows the overall system performance when running a pair of programs concurrently on a multi-core shared-memory system, using different choices for each programs grain size. The programs are implemented using Intel's TBB parallel framework, using the `parallel_for` parallel pattern, described in more detail in Section 2.3.4. Each program therefore has a grain size parameter that controls the granularity of its computation that can be tuned to improve overall system performance.

The experiment uses the BLACKSCHOLES and STREAMCLUSTER benchmarks from the PARSEC benchmark suite (Bienia, 2011). These programs are run on a 4-core shared-memory machine, and multiple repeats are used to quantify errors in the measurements. Overall system performance is measured using the ANTT and STP metrics, which are discussed in more detail in Section 2.4.1.

This experiment compares the following four approaches for choosing the grain sizes for this pair of programs:

WORST chooses the grain size that provides the worst performance, and is found using an exhaustive search of the space of possible grain sizes. This provides a lower-bound on the overall system performance.

DEFAULT uses the default grain size hard coded into each benchmark. TACHYON uses a default grain size of 1 and the default grain size of STREAMCLUSTER depends on its input and the system – it is computed by dividing the input size by the number of cores, which in this case gives a value of 1024.

ALONE-TUNED chooses the grain size for each program based on the program's performance when it is run on its own. The grain size that provides the best single-program performance for each program is used when the programs are run together. The performance of these programs against various grain sizes is shown in Figure 35. This approach is also discussed in more detail in Section 6.3.

ORACLE chooses the grain size for each program that achieves the best overall system performance. This is found using an exhaustive search of the space of possible grain size values. This provides an upper-bound on the overall system performance.

These results, shown in Figure 33 demonstrate that TBB's grain size parameter has a significant effect on performance for this pair of programs. Moreover, the default value used in the TBB library does not provide the best performance when this program pair are run together. It is also the case that choosing the best grain size for each program when they are run in isolation does not provide the best performance. This motivates the need for a more intelligent strategy for choosing the grain size for each program.

Figure 34 shows a heat map of the performance found by an exhaustive search of the space of grain size parameters, for the same pair of programs as before. The four approaches, compared previously, are also highlighted on the heat maps.

These results show that the optimisation space is complex, and that the best choice of grain size is very different to the manually chosen default that is hard-coded into TBB. Furthermore, the grain sizes chosen by the ALONE-TUNED approach are also far from the optimal grain size.

These experiments demonstrate the need for cooperatively tuning of TBB's grain size parameter, as the best choice of grainsize depends on the system workload. Moreover, the default parameter value hard coded into TBB does not provide the best system performance, nor does choosing the best grain size when each program is run on its own.

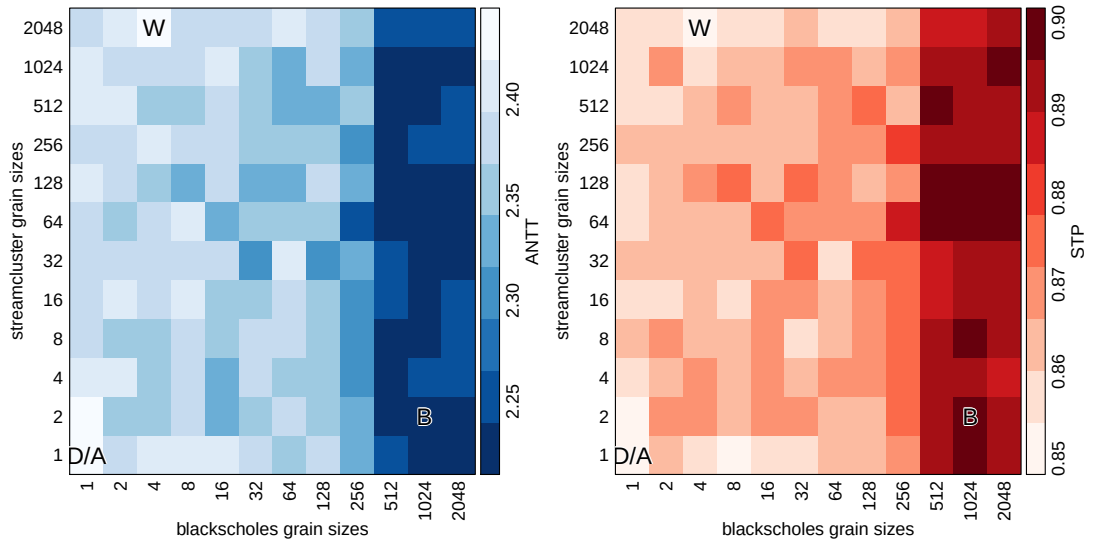

Figure 34: Heat maps showing (*i*) the Average Normalized Turnaround Time (ANTT) (left) and (*ii*) System Throughput (STP) (right) when running BLACKSCHOLES and STREAMCLUSTER at the same time on a 4-core system with different grain size settings. The x-axis shows varying grain sizes for STREAMCLUSTER and the y-axis for BLACKSCHOLES. Darker regions indicate improved performance: for ANTT lower is better, and for STP higher is better. The grain sizes chosen by the four approaches are also highlighted: (*i*) O marks the grain sizes that achieve the best possible performance found by the ORACLE approach, (*ii*) W marks the worst possible performance found by an exhaustive search, (*iii*) D marks the TBB default settings, and (*iv*) marks the point chosen by the ALONE-TUNED approach.

Table 2: The benchmark programs used in the experiments, including details of their default grain size setting and inputs used in the experiments. The grain size marked (†) is dependent on both the program input and the system.

Program	Source	Description	Default Grain Size	Inputs
blackscholes	PARSEC 3.0	Option pricing using the Black-Scholes PDE	1	50,000 options
bodytrack	PARSEC 3.0	Body tracking algorithm	8	PARSEC 'native' inputs
streamcluster	PARSEC 3.0	Online clustering algorithm	$1024^{(\dagger)}$	PARSEC 'simsmall' inputs
swaptions	PARSEC 3.0	Pricing of a portfolio of swaptions	8	500 swaptions, 500 simulations
tachyon	TBB	Parallel ray tracer	1	`820spheres.dat`

6.3 TUNING USING SINGLE-PROGRAM SCALABILITY

Figure 35 shows how program performance scales with grain size, when programs are run on their own utilising the entire machine. These experiments were run using the five benchmark programs listed in Table 2 on a 4-core shared memory system, identical to the one used for the motivational experiments in Section 6.2.

Programs differ in the best grain size. In most cases, increasing the grain size leads to worse performance. This is the case for SWAPTIONS, TACHYON and BODYTRACK. For BLACKSCHOLES, the grain size has no impact on performance, and for STREAMCLUSTER, increasing the grain size provides a slight improvement in performance.

This scalability information can be used to choose the grain size for each program running on the system. We set the grain size for each running program to be the grain size that achieves best performance when the program is run on its own on the machine. However, this tuning approach does not work in this context. On average across all the program combinations evaluated, this approach performs the same as the default TBB library settings, and therefore provides no performance benefit. These performance results are discussed in more detail in Section 6.5.3.

6.4 HEURISTIC FOR COOPERATIVELY TUNING GRAIN SIZE

In this section, we explore the correlation between TBB performance counters and system performance – measured using the ANTT and STP metrics. This is used to devise a tuning heuristic for choosing the grain size for programs when running concurrently on the same system. We later evaluate this heuristic on a profiled data set collected a priori.

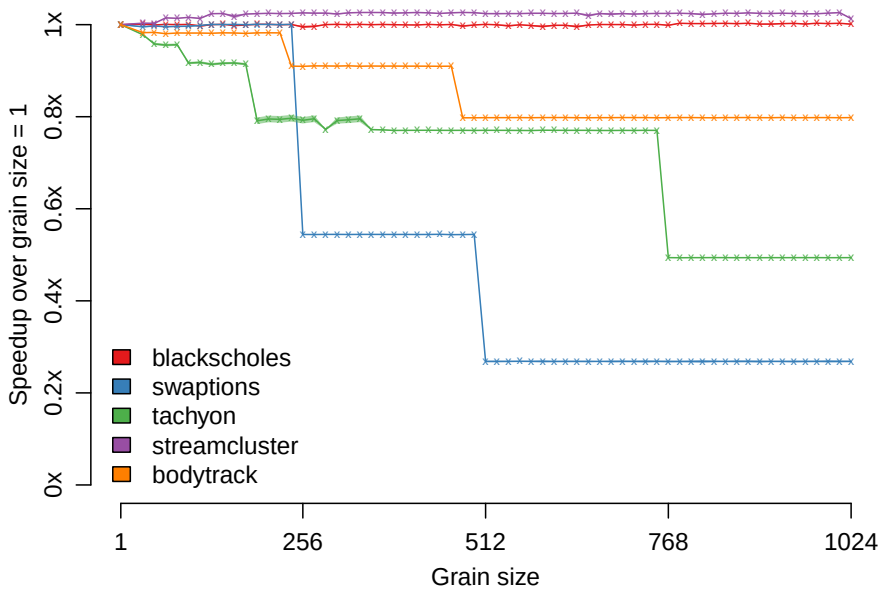

Figure 35: The performance scalability of programs with varying grain size. Each program is run on its own on a 4-core shared-memory system. The plot shows the speedup of each grain size setting, over a grain size of 1. Each program is executed for multiple repeats until the coefficient of variation dropped below 0.01. The solid lines show the average speedup across repeats, and the shaded region show a 99% confidence interval for the mean.

6.4.1 *Measuring TBB Performance Counters*

TBB includes functionality to measure the frequency of events, such as the number of TBB tasks that have been executed and the number of steals that have failed. Each TBB worker thread uses atomic increment instructions to track the frequency of these events, writing them into a table in shared memory. However, this only allows the counter frequencies to be obtained once the program has completed execution.

TBB was modified to include an additional thread that periodically writes the values of these counters to standard output, so that they can be tracked during the execution of programs. This allows a heuristic based on these performance counters to be evaluated at runtime and be used to inform the choice of the grain size.

The overheads involved in measuring these counters at runtime is evaluated in Section 6.5.4.

6.4.2 *Correlation between TBB Counters and Performance*

TBB uses a work stealing scheduler, as described in Section 2.3.4. This allows threads to steal work from one another. TBB's performance counters track the number of steals that succeed and fail. High numbers of failed steals are a good predictor of bad performance, and can be used to predict the best grain size to use.

Figure 36 shows the correlation between system performance and the number of steals failed per second, averaged across both programs. The dark blue and dark red regions show the grain sizes that achieve best performance. Cooler colours indicate low values for the number of steals failed, and these correlate with the darker, higher performance regions in the other two heatmaps.

The steals failed rate is continuously measured by TBB, unlike ANTT and STP which can only be calculated after the programs complete execution. It can therefore be used as a heuristic to guide an online adaptive approach to tune the grain size parameter. Our heuristic chooses the grain size for each program such that the average steal failure rate across programs is minimised.

6.5 EVALUATION

This section evaluates the performance found by our tuning heuristic, using a study based on profiled data collected in an exhaustive exploration of the optimisation space. The heuristic predicts the best TBB grain size using information from performance counters. It is evaluated using programs when run together, for a range different benchmark combinations.

The experimental setup is described in Section 6.5.1. Section 6.5.2 describes the exhaustive evaluation of the performance of the grain size optimisation space for different pairs of programs, used to evaluate the tuning approaches. Section 6.5.3 discusses the performance found by our tuning heuristic, based on scalability information when programs are run on their own, introduced in Section 6.3. It also compares the performance found by our heuristic against the default TBB grain size settings, against the best possible performance found using the exhaustive evaluation of the optimisation space.

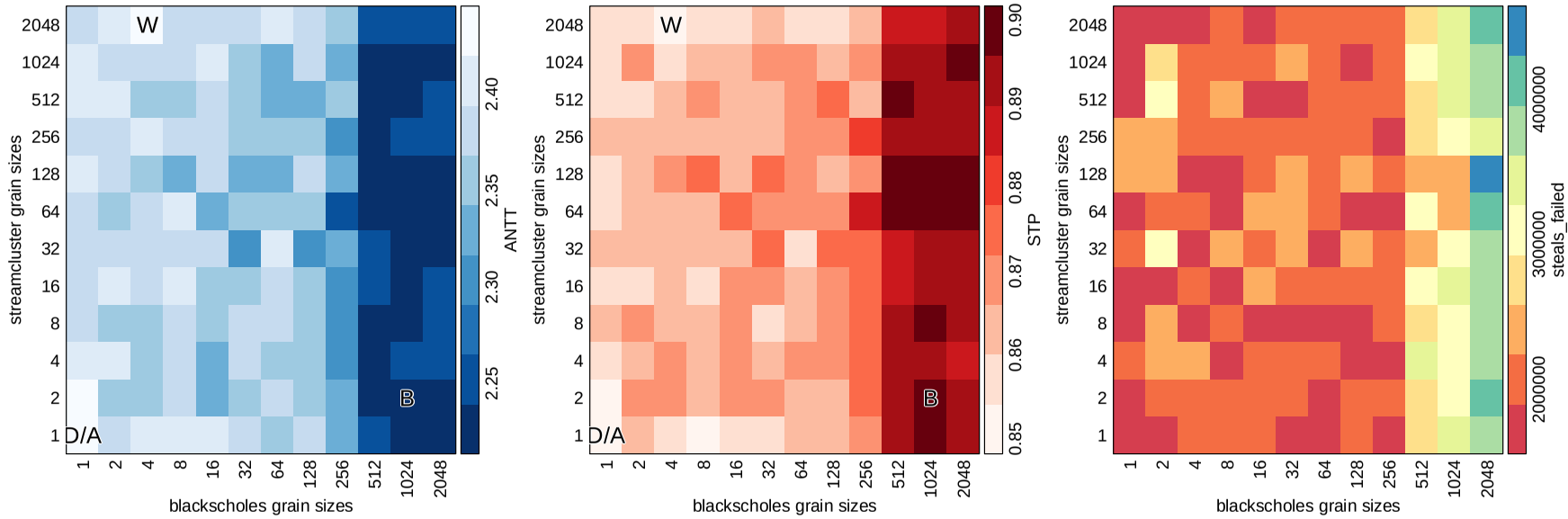

Figure 36: Heat maps showing (*i*) the Average Normalized Turnaround Time (ANTT) (left) (*ii*) System Throughput (STP) (middle) and (*ii*) steals failed (right) when running BLACKSCHOLES and STREAMCLUSTER at the same time on a 4-core system with different grain size settings. The x-axis shows varying grain sizes for STREAMCLUSTER and the y-axis for BLACKSCHOLES. Darker regions in the left two heatmaps indicate improved performance: for ANTT lower is better, and for STP higher is better. The grain sizes chosen by the four approaches are also highlighted: (*i*) O marks the grain sizes that achieve the best possible performance found by the ORACLE approach, (*ii*) W marks the worst possible performance found by an exhaustive search, (*iii*) D marks the TBB default settings, and (*iv*) marks the point chosen by the ALONE-TUNED approach.

Table 3: The program combinations used in the experiments.

#	Programs	
1	blackscholes	blackscholes
2	blackscholes	bodytrack
3	blackscholes	streamcluster
4	blackscholes	swaptions
5	blackscholes	tachyon
6	bodytrack	bodytrack
7	bodytrack	streamcluster
8	bodytrack	swaptions
9	bodytrack	tachyon
10	streamcluster	streamcluster
11	streamcluster	swaptions
12	streamcluster	tachyon
13	swaptions	swaptions
14	swaptions	tachyon
15	tachyon	tachyon

6.5.1 *Experimental Setup*

The benchmark programs used to evaluate the system are summarised in Table 2. They are all implemented using TBB's `parallel_for` pattern, and therefore contain the tunable TBB grain size parameter. They are taken from both the TBB source distribution[1] and the PARSEC 3.0 benchmark suite[2] (Bienia, 2011).

We use input sizes that require approximately half a second of execution time when run alone on the machine. The inputs used for each benchmark are summarised in Table 2. This table also shows the default grain size hard coded into each benchmark program. BLACKSCHOLES and TACHYON both use TBB's library-set default value of one. The other benchmarks have hard coded values for this parameter, chosen manually by the benchmark author. STREAMCLUSTER uses a default grain size computed using a manually devised heuristic. Its grain size is set to the size of the input data divided by the number of available cores on the system.

For the experiments we use a quad-core Intel i7-3820 processor clocked at 3.60GHz. The processor has 4 physical cores, with 2 hardware threads per core for a total of 8 hardware threads. However, we disable hyper-threading to avoid introducing noise into the experiments, using just one hardware thread per core, for a total of 4 hardware threads. The system has 16GB of main memory, and runs Linux 3.7.10. We use GCC 4.7.2 to compile the benchmark programs.

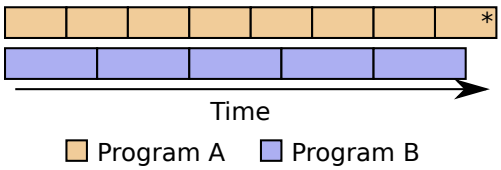

Figure 37: Diagram showing how programs are run for multiple repeats, when a pair of programs are run concurrently. The repeats are stopped once one of the two programs has run for at least 5 repeats. Performance measurements for the last execution of the shorter running program, marked $\star$, are ignored.

6.5.2 *Oracle Study*

To evaluate the different approaches for choosing values for the grain size, we exhaustively evaluate the performance at each point in the grain size parameter space, for pairs of benchmark programs. Using the set of 5 benchmark programs, we run all 15 combinations of pairs of programs. Table 1 shows the combinations, and the number used to refer to each in the graphs of results. We run an exhaustive exploration of the performance of each program combination, for a range of grain size values for each benchmark. Grain sizes of 1, 2, 4, 8, ... 2048 are explored. This provides sufficient coverage of the space, whilst keeping the space small enough so that measurements of the performance of the entire space are tractable. We also measure the performance of each benchmark combination using the default TBB grain size.

To compute the ANTT and STP for each benchmark combination, we used the execution time of each program, when run with a grain size that provides the best performance when the program is run in isolation on the system.

When running a pair of programs, each program may have very different execution times. In order to ensure that benchmark execution overlaps sufficient, we run each benchmark for at least 5 repeats. This setup is depicted in Figure 37. We measure the time for the entire execution of each program, including loading data from disk.

1 https://www.threadingbuildingblocks.org
2 http://parsec.cs.princeton.edu/overview.htm

Figure 38: Comparison of the Average Normalized Turnaround Time (ANTT) and System Throughput (STP) achieved by choosing the grain size using five different approaches, when running pairs of programs concurrently. (*i*) choosing the grain size that provides worst system performance, (*ii*) using the default value hard coded into the benchmark, (*iii*) using the best grain size from when the programs are run alone, (*iv*) choosing the grain size using our tuning heuristic and (*v*) choosing the grain size that provides the best performance, found by an exhaustive search of the space. For ANTT, lower is better, and for STP, higher is better. AVG shows the arithmetic mean across all program pairs. The program pairs are detailed in Table 3.

6.5.3 *Results*

Figure 38 shows the ANTT and STP achieved by each tuning approach for different pairs of benchmarks.

WORST shows the worst possible performance found by the exhaustive evaluation of the space of grain sizes. This provides a lower bound on the possible performance. ORACLE shows the best possible performance found in this exhaustive search, providing an upper bound on the performance that can be achieved.

DEFAULT is the performance achieved by the TBB library using its default settings. In some cases the default settings perform poorly, and in others it performs close to the oracle performance. This demonstrates that manually chosen, static parameters do not always provide the best performance. On average across all program combinations, this approach achieves an ANTT of 2.66 and STP of 0.83.

ALONE shows the performance achieved when programs are run using the grain size that provides the best performance when the program is run on its own on the system. On average across all program combinations, this approach performs similarly to DEFAULT. This poor performance demonstrates the need to tune the grain size in a cooperative manner.

TUNER achieves the best of these approaches, and is the performance found by our tuning heuristic. On average, it finds a point in the space with an ANTT of 2.35 and STP of 0.91. This approach is closest to the ORACLE performance, which is an ANTT of 2.07 and STP of 0.98.

6.5.4 *TBB Performance Counter Measurement Overheads*

Figure 39 shows the overhead involved in measuring the TBB performance counters required by the heuristic. This plot shows speedup against the time interval between measurements of the performance counters. These results show that the runtime overheads of performing this measurement are minimal, and so have a negligible effect on the overall system performance. In order to provide timely updates to the performance counters, they are sampled once every 20 milliseconds in the experiments presented here. This provides, at worst for BODYTRACK, a 0.75% performance hit, and a 0.26% performance hit on average across all of the benchmarks.

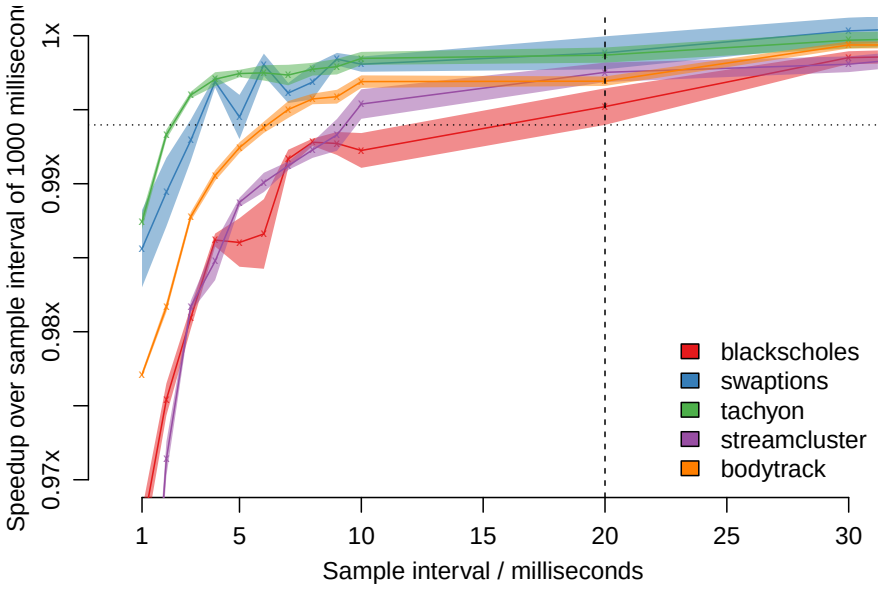

Figure 39: Diagram showing the overhead of measuring the TBB performance counters, showing the speedup relative to a sampling interval of 1 second at different values for the sampling interval. Solid lines show the mean speedup across repeated executions of the program, and the shaded regions show the 99% confidence interval for this mean. The vertical dashed line shows the sample interval at 20ms, used in the experiments, and the horizontal dotted line shows the minimum performance penalty using this sampling interval.

6.6 SUMMARY

This chapter has presented experiments that demonstrate the need for cooperative tuning of the grain size parameter in TBB. The correlation between TBB performance counters and this grain size parameter are explored, and a heuristic devised which utilises this information to choose the best grain size when programs are run concurrently. It is evaluated on a profiled data set collected a priori.

This chapter concludes the technical contributions presented by this thesis. The next chapter concludes the thesis with a brief summary of the contributions, a critical analysis of this work, and directions for future work.

7

CONCLUSIONS

This thesis has explored three challenges associated with cooperative auto-tuning of parallel programs, where the system workload is dynamically changing.

A predictive scheduler for choosing the optimal number of threads for each program running on the system was discussed in Chapter 4. Chapter 5 motivated the need for, and describes an online-adaptive scheduler for choosing where to run threads on a multi-socket machine. The challenge of tuning the grain size implementation parameter in Intel's Threading Building Blocks library is explored in Chapter 6.

The structure of this chapter is as follows. It begins in Section 7.1 with a brief summary of the contributions made by this thesis. These contributions are discussed further in Section 7.2. Section 7.3 discusses potential directions this work could be taken in in the future.

7.1 CONTRIBUTIONS

This section summarises the contributions made in the three technical chapters of this thesis, which address the three cooperative tuning challenges described in Section 1.2. Thread count tuning is performed dynamically at runtime, using scalability information collected for each program a priori. Thread counts are chosen in order to optimise the ANTT and STP performance metrics for the system as a whole. Thread placement tuning is performed using a heuristic driven scheduler. The heuristic uses memory pressure to allocate program's threads to sockets. Finally, automatically tuning the grain size implementation parameter in Intel's Threading Building Blocks is explored.

The main contributions of this thesis are:

- A technique, called ThreadTuner, for choosing thread counts for programs in the presence of dynamic system workload is developed. Thread counts are chosen based on scalability information for the programs, collected using an offline training phase. The approach dynamically adapts its decisions at runtime, in response to changes in the system workload. Runtime overheads are

reduced, by using static program information to determine when the system workload changes. Our scheduler improves overall system performance, compared to manually tuned benchmarks written using the OpenMP (Dagum and Menon, 1998) parallel framework. This work has been presented in Chapter 4.

- A heuristic and scheduler to allocate program threads to sockets is developed. The effect of different program-to-core mappings on performance is explored, revealing the performance degradation caused by ignoring the presence of separate sockets. An online adaptive scheduler that reduces interference and resource contention on shared memory multi-core multi-socket systems is developed. Its aim is to balance the pressure on the memory system of each socket, which results in improvements in system performance. This online scheduling approach is compared against two competing approaches: Callisto (Harris et al., 2014) and OpenMP (Dagum and Menon, 1998), and our approach achieves improvement in system performance. The system performance when using the dynamic scheduler is also compared to an offline static approach using the same heuristic. It is shown that the dynamic approach improves system performance. This work has been presented in Chapter 5.

- The performance impact of the grain size implementation parameter in Intel's Threading Building Blocks is investigated. An oracle study of the optimisation space is performed, comparing the best, worst and default performance achieved when varying TBB's grain size parameter. A heuristic is devised from this data, which uses TBB performance statistics to tune the grain size parameter. This work has been presented in Chapter 6.

7.2 CRITICAL ANALYSIS

This thesis has provided some significant contributions to the field of cooperative auto-tuning. This section discusses how these contributions fit into the wider context, provides analysis of the pros and cons of the approaches developed in this thesis and discusses some of the pitfalls encountered during this work. Various challenges of cooperative auto-tuning not addressed by this thesis are discussed in future work Section 7.3.1.

7.2.1 *Benchmark Suites*

A custom skeleton library was developed for the thread count tuning presented in Chapter 4, and the intention was to use this library for the remaining chapters. However, a benchmark suite also needed to be created for the library. Implementing such a benchmark suite would have required a large programming effort beyond the scope of this thesis. Even if a new benchmark suite had been devised, the wider community would not necessarily be convinced that such a suite would represent a sensible set of applications. Moreover, performance comparison with existing systems would be difficult using a completely different benchmark set. We therefore used OpenMP and TBB for the later technical chapters as these parallel libraries are more established and have a wide range of available benchmark programs.

7.2.2 *Trained Approaches*

The approach used in Chapter 4 relies on an offline training phase. In contrast to a heuristic driven online approach, this is expensive in terms of computation time. Training is also unlikely to be platform independent.

The training is also unlikely to be agnostic to the inputs that a program is given. In our experiments, programs are given identical inputs each time they are run, which is not representative of the real world.

Training also requires a set of benchmarks that provides an accurate representation of the types of program that will be run in production. This is a disadvantage when the approach needs to be trained for use in a general purpose environment where the mix of applications is unknown.

However, if done correctly, training provides accurate a priori information from which tuning decisions can be made. It would be impossible to make tuning decisions without some form of information. It is possible that a heuristic exists which is as effective as this offline trained approach, however none exists that I am aware of.

7.2.3 *Scaling to Larger Systems*

It is not clear whether LIRA, our thread placement tuner described in Chapter 5, scales to larger systems and larger sets of applications. The heuristic formula can be extended to accommodate more concurrently running applications, and the intu-

ition behind maintaining balanced memory pressure across sockets suggests that the approach should scale to larger systems. However, our approach does not consider, for example, the overhead of migrating threads between sockets. As the number of applications and sockets grows this overhead may become an issue.

7.2.4 *Program Transformation*

This work tunes programs by adjusting parameters of the program's implementation. Other works transform the program itself in order to improve performance (Ansel et al., 2009), however there is no previous work on doing this in a cooperative manner on a shared system.

The performance gains available from modifying the program source code are likely large, as this provides a larger scope for program modification beyond adjusting implementation parameters. However, for such an approach to be tractable the scope of program modifications would need to be constrained. These constraints would then define the optimisation space over which we need to tune.

7.3 FUTURE WORK

This section discusses potential directions in which this work could be extended.

7.3.1 *Additional Challenges for Cooperative Tuning*

Cooperatively tuning applications is not isolated to the three challenges addressed in this thesis, although they do cover a significant portion of the tuning problem. The following outlines other potential challenges for cooperative tuning on multi-core systems:

SCHEDULING THREADS IN TIME The work in this thesis relies on there only being one thread scheduled to each core. If this is not the case, then one approach is to multiplex the core among multiple threads, by context switching between them. This would need to be done in a manner that takes cooperative tuning into account.

ADAPTING THE SKELETON IMPLEMENTATION There are often many different algorithms that can be used to implement a skeleton. Sorting is a clear example: many sorting algorithms exist, with different tradeoffs, and the choice of algorithm is likely to have an impact on programs cooperation for resources on shared systems.

ADAPTING DATA STRUCTURES The choice of data structure is another high-level implementation decision that can be varies. Different choices of data structures, with different tradeoffs, will have an impact on cooperation between programs for resources. For example, some data structures may be more space efficient than others – which may be beneficial when sharing parts of the cache hierarchy – but may require more computational resource to use.

7.3.2 *Interaction between the Three Challenges*

Another area for future work is to address the ways in which the three cooperative tuning challenges explored in this thesis interact.

For example, the choice of high level skeleton parameters – such as TBB's grain size explored in Chapter 6 – is likely to have an impact on the choice of the number of threads. If the system is heavily over-subscribed, using fewer threads will likely improve system performance, and also reducing the grain size so that fewer tasks are present on the system for the reduced number of threads to execute.

The techniques used to address these techniques may also be complimentary, but this needs to be explored further. For example, the out cooperative thread count tuning technique could be used within sockets, where the program to socket mapping is chosen by our cooperative thread placement auto-tuning technique.

7.3.3 *Over-subscription and Simultaneous Multi-threading*

The thread count auto-tuning presented in this thesis ignores the potential performance improvements that could be gained by exploited Simultaneous Multi-Threading (SMT), also known as Hyper-Threading or by spawning more threads than there are cores. In some contexts, executing more threads than there are cores can be beneficial, as the threads execution can be overlapped to hide stalls in execution caused by I/O or synchronisation.

7.3.4 *Wider Range of Workloads*

The thread placement tuner is only evaluated for 4 concurrently running programs, for tractability reasons. Extending this analysis to larger numbers of applications should demonstrate its effectiveness for more diverse workloads. It could also be extended to use of finer grained scheduling, by co-scheduling individual threads, instead of entire programs of threads.

7.4 SUMMARY

This chapter concludes the thesis. It has provided a summary of the main contributions made by this work, a critical analysis of the material presented and discusses ideas for future work in the field of cooperatively tuning parallel programs.

BIBLIOGRAPHY

Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael O'Boyle, John Thomson, Marc Toussaint, and Chris K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4th Symposium on Code Generation and Optimization*, pages 295–305, 2006. (Cited on page 19)

Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proceedings of the 11th Conference on Parallel and Distributed Computing and Systems*, pages 955–962, 1999. (Cited on page 19)

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, chapter 13. Wiley, 2013. (Cited on page 52)

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th Conference on Programming Language Design and Implementation*, 2009. (Cited on pages 2, 53 and 122)

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on pages 1 and 3)

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The Journal of Supercomputer Applications, 1991. (Cited on pages 58, 60, 65 and 76)

Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: A novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the Conference on Supercomputing*, 2008. (Cited on pages 43, 44 and 51)

Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th Conference on Supercomputing*, pages 189–199. ACM, 2010. (Cited on pages 37 and 43)

Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011. (Cited on pages 107 and 114)

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. (Cited on pages 18 and 19)

François Bodin, Toru Kisuki, Peter M. W. Knijnenburg, Michael O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the 1st Workshop on Profile Directed Feedback-Compilation*, 1998. (Cited on page 17)

Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly, 2008. (Cited on pages 57, 62 and 77)

Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001. (Cited on page 97)

Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 20th Conference on Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE Computer Society, 2011. (Cited on pages 53 and 55)

James R. Bulpin and Ian A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2005. (Cited on page 49)

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*, pages 47–56, 2011. (Cited on page 54)

Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 35–46. ACM, 2011. (Cited on pages 53 and 55)

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th*

Symposium on High-Performance Computer Architecture, pages 340–351, 2005. (Cited on page 50)

Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 25th Parallel Distributed Processing Symposium*, pages 676–687, 2011. (Cited on page 2)

Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. (Cited on pages 1 and 21)

Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004. (Cited on page 1)

Alexander Collins, Christian Fensch, and Hugh Leather. Auto-tuning parallel skeletons. *Parallel Processing Letters*, 22(2):16, 2012. (Cited on page 52)

Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 57–66, 2008. (Cited on page 24)

Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000. (Cited on page 37)

Julita Corbalán, Xavier Martorell, and Jesús Labarta. Improving gang scheduling through job performance analysis and malleability. In *Proceedings of the 15th Conference on Supercomputing*, pages 303–311, 2001. (Cited on page 45)

Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computing Science and Engineering*, 5(1):46–55, 1998. (Cited on pages 2, 8, 23, 57 and 120)

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–394. ACM, 2013. (Cited on pages 46 and 51)

Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In

Proceedings of the 4th Workshop on Multicore Software Engineering, pages 25–32, 2011. (Cited on pages 2, 19, 54 and 55)

Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the SkePU skeleton programming library. In *Advanced Parallel Processing Technologies*, volume 8299, pages 170–183. 2013. (Cited on pages 54 and 55)

Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4), 2013. (Cited on pages 45 and 51)

Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. vGreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the 14th Symposium on Low Power Electronics and Design*. ACM, 2009. (Cited on page 49)

Lieven Eeckhout. *Computer Architecture Performance Evaluation Methods*, volume 10. Morgan & Claypool, 2010. (Cited on pages 26 and 27)

Michael O'Boyle Murali Krishna Emani. Change detection based parallelism mapping: Exploiting offline models and online adaptation. In *Proceedings of the 27th Workshop on Languages and Compilers for Parallel Computing*, 2014. (Cited on page 41)

Zheng Wang Murali Krishna Emani and Michael O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Proceedings of the 11th Symposium on Code Generation and Optimization*, 2013. (Cited on pages 40 and 41)

Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th Conference on Parallel Architecture and Compilation Techniques*. IEEE, 2007. (Cited on page 44)

D. Feinberg. Database management systems technology trends. Technical report, 2006. (Cited on page 3)

J. A. Fisher. Walk-time techniques: catalyst for architectural change. *Computer*, 30(9): 40–42, 1997. (Cited on page 53)

Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel Distributed Systems*, 16(11):1066–1077, 2005. (Cited on page 47)

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Conference on Object-oriented Programming Systems and Applications*, pages 57–76, 2007. (Cited on page 31)

Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010. (Cited on pages 2, 21, 35 and 52)

P. Greenhalgh. big.LITTLE processing with arm cortex-a15 & cortex-a7, 2011. (Cited on page 15)

Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th Conference on High Performance and Embedded Architectures and Compilers*, pages 117–126. ACM, 2011. (Cited on pages 40 and 41)

Tim Harris, Yossi Lev, Victor Luchangco, Virendra Marathe, and Mark Moir. Constrained data-driven parallelism. In *Proceedings of the 5th Workshop on Hot Topics in Parallelism*, 2013. (Cited on page 97)

Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the 9th ACM European Conference on Computer Systems*, 2014. (Cited on pages 8, 42, 83, 84, 86, 90, 104 and 120)

Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly, 2013. (Cited on page 76)

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362. ACM, 2012. (Cited on pages 85 and 97)

Peter J. Huber. *Robust Statistics*. John Wiley and Sons Inc., 2005. (Cited on page 32)

Intel. An introduction to the intel quickpath interconnect, 2009. (Cited on page 6)

Intel. Intel Threading Building Blocks, 2012. http://software.intel.com/en-us/articles/intel-tbb/. (Cited on pages 2, 4, 7, 10, 24, 105 and 106)

James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013. (Cited on page 16)

Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008. (Cited on page 51)

Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. Adaptive execution techniques for SMT multiprocessor architectures. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pages 236–246. ACM, 2005. (Cited on pages 54 and 55)

C. Kim, J.H. Jung, T.K. Ko, S.W. Lim, S. Kim, K. Lee, and W. Lee. MobileBench: A thorough performance evaluation framework for mobile systems. In *Proceedings of the 1st International Workshop on Parallelism in Mobile Platforms*, 2013. (Cited on page 3)

Toru Kisuki, Peter M. W. Knijnenburg, and Michael O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 19th Conference on Parallel Architectures and Compilation Techniques*, page 237, 2000. (Cited on page 17)

Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. autopin – automated optimization of thread-to-core pinning on multicore systems. In *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590, pages 219–235. Springer Berlin / Heidelberg, 2011. (Cited on page 48)

Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008. (Cited on pages 44 and 51)

Alexey Kukanov and Michael Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 2007. (Cited on page 24)

Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th Symposium on Code Generation and Optimization*, pages 81–91, 2009. (Cited on page 20)

Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread Tailor: Dynamically weaving threads together for efficient, adaptive par-

allel applications. In *Proceedings of the 37th Symposium on Computer Architecture*, pages 270–279, 2010. (Cited on pages 39 and 41)

Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms. In *Proceedings of the PARMA-DITAM Workshop*. ACM, 2014. (Cited on pages 45 and 51)

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th Symposium on High-Performance Computer Architecture*, pages 367–378, 2008. (Cited on pages 46 and 51)

Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. (Cited on page 18)

Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Symposium on Code Generation and Optimization*. ACM, 2010. (Cited on pages 47 and 51)

Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012. (Cited on page 21)

Robert L. McGregor, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proceedings of the 19th Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005. (Cited on pages 44 and 51)

Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 2010. (Cited on page 49)

D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, 2009. (Cited on page 6)

C. Moore. Data processing in exascale-class computer systems. In *Proceedings of the Salishan Conference on High Speed Computing*, 2011. (Cited on page 14)

R. W. Moore and B. R. Childers. Using utility prediction models to dynamically choose program thread counts. In *Proceedings of the IEEE Symposium on Performance Analysis of Systems and Software*, pages 135–144, 2012. (Cited on pages 38 and 41)

Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defence HPCMP Users Group Conference*, pages 7–10, 1999. (Cited on pages 28 and 88)

Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 2010. (Cited on page 50)

Nvidia. Thrust, 2012. http://thrust.github.com. (Cited on pages 54 and 55)

OpenMP, 2015. http://www.openmp.org. (Cited on pages 2 and 23)

David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990. (Cited on page 25)

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. ADAPT: A framework for coscheduling multithreaded programs. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013. (Cited on page 37)

Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Symposium on Microarchitecture*. IEEE, 2006. (Cited on page 50)

Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, pages 133–144, 2012. (Cited on page 40)

John Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 1995. (Cited on page 32)

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the 9th Conference on Generative programming and component engineering*, pages 127–136. ACM, 2010. (Cited on page 53)

Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st Conference on Parallel Architectures and Compilation Techniques*, pages 107–116, 2012. (Cited on pages 36, 38 and 41)

Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2000. (Cited on page 49)

G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2002. (Cited on page 50)

Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7): 54–62, 2005. (Cited on pages 1 and 14)

Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th Symposium on Computer Architecture*. ACM, 2011. (Cited on pages 47 and 51)

John D. Thomson. *Auto-tuning Performance on Multicore Computers*. PhD thesis, Institute of Computing Systems Architecture, Informatics Department, University of Edinburgh, 2009. (Cited on page 19)

Kai Tian, Yunlian Jiang, and Xipeng Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the 6th Conference on Computing Frontiers*. ACM, 2009. (Cited on page 45)

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the 1st Symposium on Code Generation and Optimization*, pages 204–215, 2003. (Cited on pages 18 and 19)

Zheng Wang and Michael F. P. O'Boyle. Partitioning streaming parallelism for multicores: A machine learning based approach. In *Proceedings of the 19th Conference on Parallel Architectures and Compilation Techniques*, pages 307–318, 2010. (Cited on pages 2 and 39)

Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008. (Cited on page 45)

Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceed-*

ings of the 8th ACM European Conference on Computer Systems. ACM, 2013. (Cited on page 50)

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010. (Cited on pages 43 and 51)

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1), 2012. (Cited on pages 35, 42 and 52)